ARM A64 Instruction Set Architecture
ARMv8, for ARMv8-A architecture profile

Copyright © 2010-2018 Arm Limited (or its affiliates). All rights reserved.

Release Information

For information on the change history and known issues for this release, see the Release Notes in the A64 ISA XML for ARMv8.5 (00bet8).

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with or are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the Arm’s trademark usage guidelines http://www.arm.com/company/policies/trademarks.

Copyright © 2013-2018 Arm Limited (or its affiliates). All rights reserved.

110 Fulbourn Road, Cambridge, England CB1 9NJ.
LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is for a Beta product, that is a product under development.

Web Address

http://www.arm.com
**A64 -- Base Instructions (alphabetic order)**

**ADC**: Add with Carry.

**ADCS**: Add with Carry, setting flags.

**ADD (extended register)**: Add (extended register).

**ADD (immediate)**: Add (immediate).

**ADD (shifted register)**: Add (shifted register).

**ADDG**: Add with Tag.

**ADDS (extended register)**: Add (extended register), setting flags.

**ADDS (immediate)**: Add (immediate), setting flags.

**ADDS (shifted register)**: Add (shifted register), setting flags.

**ADR**: Form PC-relative address.

**ADRP**: Form PC-relative address to 4KB page.

**AND (immediate)**: Bitwise AND (immediate).

**AND (shifted register)**: Bitwise AND (shifted register).

**ANDS (immediate)**: Bitwise AND (immediate), setting flags.

**ANDS (shifted register)**: Bitwise AND (shifted register), setting flags.

**ASR (immediate)**: Arithmetic Shift Right (immediate): an alias of SBFM.

**ASR (register)**: Arithmetic Shift Right (register): an alias of ASRV.

**ASRV**: Arithmetic Shift Right Variable.

**AT**: Address Translate: an alias of SYS.

**AUTDA, AUTDZA**: Authenticate Data address, using key A.

**AUTDB, AUTDZB**: Authenticate Data address, using key B.

**AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA**: Authenticate Instruction address, using key A.

**AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB**: Authenticate Instruction address, using key B.

**AXFlag**: Convert floating-point condition flags from ARM to external format.

**B**: Branch.

**B.cond**: Branch conditionally.

**BFC**: Bitfield Clear: an alias of BFM.

**BFI**: Bitfield Insert: an alias of BFM.

**BFM**: Bitfield Move.

**BFXIL**: Bitfield extract and insert at low end: an alias of BFM.

**BIC (shifted register)**: Bitwise Bit Clear (shifted register).

**BICS (shifted register)**: Bitwise Bit Clear (shifted register), setting flags.

**BL**: Branch with Link.

**BLR**: Branch with Link to Register.
BLRAA, BLRAAZ, BLRAB, BLRABZ: Branch with Link to Register, with pointer authentication.
BR: Branch to Register.
BRAA, BRAAZ, BRAB, BRABZ: Branch to Register, with pointer authentication.
BRK: Breakpoint instruction.
BTI: Branch Target Identification.
CAS, CASA, CASAL, CASL: Compare and Swap word or doubleword in memory.
CASB, CASAB, CASALB, CASLB: Compare and Swap byte in memory.
CASH, CASAH, CASALH, CASLH: Compare and Swap halfword in memory.
CASP, CASPA, CASPAL, CASPL: Compare and Swap Pair of words or doublewords in memory.
CBNZ: Compare and Branch on Nonzero.
CBZ: Compare and Branch on Zero.
CCMN (immediate): Conditional Compare Negative (immediate).
CCMN (register): Conditional Compare Negative (register).
CCMP (immediate): Conditional Compare (immediate).
CCMP (register): Conditional Compare (register).
CFINV: Invert Carry Flag.
CFP: Control Flow Prediction Restriction by Context: an alias of SYS.
CINC: Conditional Increment: an alias of CSINC.
CINV: Conditional Invert: an alias of CSINV.
CLREX: Clear Exclusive.
CLS: Count Leading Sign bits.
CLZ: Count Leading Zeros.
CMN (immediate): Compare Negative (immediate): an alias of ADDS (immediate).
CMP (immediate): Compare (immediate): an alias of SUBS (immediate).
CMPP: Compare with Tag: an alias of SUBPS.
CNEG: Conditional Negate: an alias of CSNEG.
CPP: Cache Prefetch Prediction Restriction by Context: an alias of SYS.
CRC32CB, CRC32CH, CRC32CW, CRC32CX: CRC32C checksum.
CSDB: Consumption of Speculative Data Barrier.
CSEL: Conditional Select.
CSET: Conditional Set: an alias of CSINC.
CSETM: Conditional Set Mask: an alias of CSINV.
CSINC: Conditional Select Increment.
CSINV: Conditional Select Invert.
CSNEG: Conditional Select Negation.
DC: Data Cache operation: an alias of SYS.
DCPS1: Debug Change PE State to EL1..
DCPS2: Debug Change PE State to EL2..
DCPS3: Debug Change PE State to EL3.
DMB: Data Memory Barrier.
DRPS: Debug restore process state.
DSB: Data Synchronization Barrier.
DVP: Data Value Prediction Restriction by Context: an alias of SYS.
EON (shifted register): Bitwise Exclusive OR NOT (shifted register).
EOR (immediate): Bitwise Exclusive OR (immediate).
EOR (shifted register): Bitwise Exclusive OR (shifted register).
ERET: Exception Return.
ERETAA, ERETAB: Exception Return, with pointer authentication.
FSB: Error Synchronization Barrier.
EXTR: Extract register.
GMI: Tag Mask Insert.
HINT: Hint instruction.
HLT: Halt instruction.
HVC: Hypervisor Call.
IC: Instruction Cache operation: an alias of SYS.
IRG: Insert Random Tag.
ISB: Instruction Synchronization Barrier.
LDADD, LDADDA, LDADDAL, LDADDL: Atomic add on word or doubleword in memory.
LDADDB, LDADDA, LDADDALB, LDADDLB: Atomic add on byte in memory.
LDADDH, LDADDAH, LDADDALH, LDADDLH: Atomic add on halfword in memory.
LDAPR: Load-Acquire RCpc Register.
LDAPRB: Load-Acquire RCpc Register Byte.
LDAPRH: Load-Acquire RCpc Register Halfword.
LDAPUR: Load-Acquire RCpc Register (unscaled).
LDAPURB: Load-Acquire RCpc Register Byte (unscaled).
LDAPURH: Load-Acquire RCpc Register Halfword (unscaled).
LDAPURSB: Load-Acquire RCpc Register Signed Byte (unscaled).
LDAPURSH: Load-Acquire RCpc Register Signed Halfword (unscaled).
LDAPURSW: Load-Acquire RCpc Register Signed Word (unscaled).
LDAR: Load-Acquire Register.
LDARB: Load-Acquire Register Byte.
LDARH: Load-Acquire Register Halfword.
LDAXP: Load-Acquire Exclusive Pair of Registers.
LDAXR: Load-Acquire Exclusive Register.
LDAXRB: Load-Acquire Exclusive Register Byte.
LDAXRH: Load-Acquire Exclusive Register Halfword.
LDCLR, LDCLRA, LDCLRAL, LDCLRL: Atomic bit clear on word or doubleword in memory.
LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB: Atomic bit clear on byte in memory.
LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH: Atomic bit clear on halfword in memory.
LDEOR, LDEORA, LDEORAL, LDEORL: Atomic exclusive OR on word or doubleword in memory.
LDEORB, LDEORAB, LDEORALB, LDEORLB: Atomic exclusive OR on byte in memory.
LDEORH, LDEORAH, LDEORALH, LDEORLH: Atomic exclusive OR on halfword in memory.
LDG: Load Allocation Tag.
LDGV: Load Allocation Tag.
LDLAR: Load LOAcquire Register.
LDLARB: Load LOAcquire Register Byte.
LDLARH: Load LOAcquire Register Halfword.
LDNP: Load Pair of Registers, with non-temporal hint.
LDP: Load Pair of Registers.
LDPSW: Load Pair of Registers Signed Word.
LDR (immediate): Load Register (immediate).
LDR (literal): Load Register (literal).
LDR (register): Load Register (register).
LDRAA, LDRAB: Load Register, with pointer authentication.
LDRB (immediate): Load Register Byte (immediate).
LDRB (register): Load Register Byte (register).
LDRH (immediate): Load Register Halfword (immediate).
LDRH (register): Load Register Halfword (register).
LDRSB (immediate): Load Register Signed Byte (immediate).
LDRSB (register): Load Register Signed Byte (register).
LDRSH (immediate): Load Register Signed Halfword (immediate).
LDRSH (register): Load Register Signed Halfword (register).
LDRSW (immediate): Load Register Signed Word (immediate).
LDRSW (literal): Load Register Signed Word (literal).
LDRSW (register): Load Register Signed Word (register).
LDSET, LDSETA, LDSETAL, LDSETL: Atomic bit set on word or doubleword in memory.
LDSETB, LDSETAB, LDSETALB, LDSETLB: Atomic bit set on byte in memory.
LDSETH, LDSETAH, LDSETALH, LDSETLH: Atomic bit set on halfword in memory.
LDMAX, LDSMAXA, LDSMAXAL, LDSMAXL: Atomic signed maximum on word or doubleword in memory.
LDMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB: Atomic signed maximum on byte in memory.
LDMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH: Atomic signed maximum on halfword in memory.
LDMIN, LDSMINA, LDSMINAL, LDSMINL: Atomic signed minimum on word or doubleword in memory.
LDMINB, LDSMINAB, LDSMINALB, LDSMINLB: Atomic signed minimum on byte in memory.
LDMINH, LDSMINAH, LDSMINALH, LDSMINLH: Atomic signed minimum on halfword in memory.
LDTR: Load Register (unprivileged).
LDTRB: Load Register Byte (unprivileged).
LDTRH: Load Register Halfword (unprivileged).
LDTRSB: Load Register Signed Byte (unprivileged).
LDTRSH: Load Register Signed Halfword (unprivileged).
LDTRSW: Load Register Signed Word (unprivileged).
LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL: Atomic unsigned maximum on word or doubleword in memory.
LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB: Atomic unsigned maximum on byte in memory.
LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH: Atomic unsigned maximum on halfword in memory.
LDUMIN, LDUMINA, LDUMINAL, LDUMINL: Atomic unsigned minimum on word or doubleword in memory.
LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB: Atomic unsigned minimum on byte in memory.
LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH: Atomic unsigned minimum on halfword in memory.
LDUR: Load Register (unscaled).
LDURB: Load Register Byte (unscaled).
LDURH: Load Register Halfword (unscaled).
LDURSB: Load Register Signed Byte (unscaled).
LDURSH: Load Register Signed Halfword (unscaled).
LDURSW: Load Register Signed Word (unscaled).
LDXP: Load Exclusive Pair of Registers.
LDXR: Load Exclusive Register.
LDXRB: Load Exclusive Register Byte.
LDXRH: Load Exclusive Register Halfword.
LSL (immediate): Logical Shift Left (immediate): an alias of UBFM.
LSL (register): Logical Shift Left (register): an alias of LSLV.
LSLV: Logical Shift Left Variable.
LSR (immediate): Logical Shift Right (immediate): an alias of UBFM.

LSR (register): Logical Shift Right (register): an alias of LSRV.

LSRV: Logical Shift Right Variable.

MADD: Multiply-Add.

MNEG: Multiply-Negate: an alias of MSUB.

MOV (bitmask immediate): Move (bitmask immediate): an alias of ORR (immediate).

MOV (inverted wide immediate): Move (inverted wide immediate): an alias of MOVN.


MOV (to/from SP): Move between register and stack pointer: an alias of ADD (immediate).

MOV (wide immediate): Move (wide immediate): an alias of MOVZ.

MOVK: Move wide with keep.

MOVN: Move wide with NOT.

MOVZ: Move wide with zero.

MRS: Move System Register.

MSR (immediate): Move immediate value to Special Register.

MSR (register): Move general-purpose register to System Register.

MSUB: Multiply-Subtract.

MUL: Multiply: an alias of MADD.

MVN: Bitwise NOT: an alias of ORN (shifted register).


NEGS: Negate, setting flags: an alias of SUBS (shifted register).

NGC: Negate with Carry: an alias of SBC.

NGCS: Negate with Carry, setting flags: an alias of SBCS.

NOP: No Operation.

ORN (shifted register): Bitwise OR NOT (shifted register).

ORR (immediate): Bitwise OR (immediate).

ORR (shifted register): Bitwise OR (shifted register).

PACDA, PACDZA: Pointer Authentication Code for Data address, using key A.

PACDB, PACDZB: Pointer Authentication Code for Data address, using key B.

PACGA: Pointer Authentication Code, using Generic key.

PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA: Pointer Authentication Code for Instruction address, using key A.

PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB: Pointer Authentication Code for Instruction address, using key B.

PRFM (immediate): Prefetch Memory (immediate).

PRFM (literal): Prefetch Memory (literal).

PRFM (register): Prefetch Memory (register).

PRFM (unscaled offset): Prefetch Memory (unscaled offset).
PSB CSYNC: Profiling Synchronization Barrier.
PSSBB: Physical Speculative Store Bypass Barrier.
RBIT: Reverse Bits.
RET: Return from subroutine.
RETTA, RETAB: Return from subroutine, with pointer authentication.
REV: Reverse Bytes.
REV16: Reverse bytes in 16-bit halfwords.
REV32: Reverse bytes in 32-bit words.
REV64: Reverse Bytes: an alias of REV.
RMIF: Rotate, Mask Insert Flags.
ROR (immediate): Rotate right (immediate): an alias of EXTR.
ROR (register): Rotate Right (register): an alias of RORV.
RORV: Rotate Right Variable.
SB: Speculation Barrier.
SBC: Subtract with Carry.
SBCS: Subtract with Carry, setting flags.
SBFIZ: Signed Bitfield Insert in Zero: an alias of SBFM.
SBFM: Signed Bitfield Move.
SBFX: Signed Bitfield Extract: an alias of SBFM.
SDIV: Signed Divide.
SETF8, SETF16: Evaluation of 8 or 16 bit flag values.
SEV: Send Event.
SEVL: Send Event Local.
SMADDL: Signed Multiply-Add Long.
SMC: Secure Monitor Call.
SMNEGL: Signed Multiply-Negate Long: an alias of SMSUBL.
SMSUBL: Signed Multiply-Subtract Long.
SMULH: Signed Multiply High.
SMULL: Signed Multiply Long: an alias of SMADDL.
SSBB: Speculative Store Bypass Barrier.
ST2G: Store Allocation Tags.
STADD, STADDL: Atomic add on word or doubleword in memory, without return: an alias of LDADD, LDADDA, LDADDAL, LDADDL.
STADDB, STADDLB: Atomic add on byte in memory, without return: an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB.
STADDH, STADDLH: Atomic add on halfword in memory, without return: an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH.
STCLR, STCLRL: Atomic bit clear on word or doubleword in memory, without return: an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
STCLRB, STCLRLB: Atomic bit clear on byte in memory, without return: an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.
STCLRH, STCLRLH: Atomic bit clear on halfword in memory, without return: an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.

STEO, STEORL: Atomic exclusive OR on word or doubleword in memory, without return: an alias of LDEOR, LDEORA, LDEORAL, LDEORL.

STEB, STEORLB: Atomic exclusive OR on byte in memory, without return: an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB.

STEOH, STEORLH: Atomic exclusive OR on halfword in memory, without return: an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH.

STG: Store Allocation Tag.

STGP: Store Allocation Tag and Pair of registers.

STGV: Store Tag Vector.

STLLR: Store LORelease Register.

STLLRB: Store LORelease Register Byte.

STLLRH: Store LORelease Register Halfword.

STLR: Store-Release Register.

STLRB: Store-Release Register Byte.

STLRH: Store-Release Register Halfword.

STLUR: Store-Release Register (unscaled).

STLURB: Store-Release Register Byte (unscaled).

STLURH: Store-Release Register Halfword (unscaled).

STLP: Store Pair of Registers.

STR (immediate): Store Register (immediate).

STR (register): Store Register (register).

STRB (immediate): Store Register Byte (immediate).

STRB (register): Store Register Byte (register).

STRH (immediate): Store Register Halfword (immediate).

STRH (register): Store Register Halfword (register).

STSET, STSETL: Atomic bit set on word or doubleword in memory, without return: an alias of LDSET, LDSETA, LDSEtal, LDSEtl.

STSETB, STSETLB: Atomic bit set on byte in memory, without return: an alias of LDSETB, LDSETAB, LDSEtALB, LDSEtLB.

STSETH, STSETLH: Atomic bit set on halfword in memory, without return: an alias of LDSETH, LDSETAH, LDSEtalH, LDSEtLH.

STSMAX, STSMAXL: Atomic signed maximum on word or doubleword in memory, without return: an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXAL.

STSMAXB, STSMAXLB: Atomic signed maximum on byte in memory, without return: an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.

STSMAXH, STSMAXLH: Atomic signed maximum on halfword in memory, without return: an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
STSMIN, STSMINL: Atomic signed minimum on word or doubleword in memory, without return: an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.

STSMINB, STSMINLB: Atomic signed minimum on byte in memory, without return: an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.

STSMINH, STSMINLH: Atomic signed minimum on halfword in memory, without return: an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.

STTR: Store Register (unprivileged).

STTRB: Store Register Byte (unprivileged).

STTRH: Store Register Halfword (unprivileged).

STUMAX, STUMAXL: Atomic unsigned maximum on word or doubleword in memory, without return: an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.

STUMAXB, STUMAXLB: Atomic unsigned maximum on byte in memory, without return: an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.

STUMAXH, STUMAXLH: Atomic unsigned maximum on halfword in memory, without return: an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.

STUMIN, STUMINL: Atomic unsigned minimum on word or doubleword in memory, without return: an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.

STUMINB, STUMINLB: Atomic unsigned minimum on byte in memory, without return: an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.

STUMINH, STUMINLH: Atomic unsigned minimum on halfword in memory, without return: an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.

STUR: Store Register (unscaled).

STURB: Store Register Byte (unscaled).

STURH: Store Register Halfword (unscaled).

STXP: Store Exclusive Pair of registers.

STXR: Store Exclusive Register.

STXRB: Store Exclusive Register Byte.

STXRH: Store Exclusive Register Halfword.

STZ2G: Store Allocation Tags, Zeroing.

STZG: Store Allocation Tag, Zeroing.

SUB (extended register): Subtract (extended register).

SUB (immediate): Subtract (immediate).

SUB (shifted register): Subtract (shifted register).

SUBG: Subtract with Tag.

SUBP: Subtract Pointer.

SUBPS: Subtract Pointer, setting Flags.

SUBS (extended register): Subtract (extended register), setting flags.

SUBS (immediate): Subtract (immediate), setting flags.

SUBS (shifted register): Subtract (shifted register), setting flags.

SVC: Supervisor Call.

SWP, SWPA, SWPAL, SWPL: Swap word or doubleword in memory.
**A64 -- Base Instructions (alphabetic order)**

**SWPB, SWPAB, SWPALB, SWPLB**: Swap byte in memory.

**SWPH, SWPAH, SWPALH, SWPLH**: Swap halfword in memory.

**SXTB**: Signed Extend Byte: an alias of SBFM.

**SXTH**: Sign Extend Halfword: an alias of SBFM.

**SXTW**: Sign Extend Word: an alias of SBFM.

**SYS**: System instruction.

**SYSL**: System instruction with result.

**TBNZ**: Test bit and Branch if Nonzero.

**TBZ**: Test bit and Branch if Zero.

**TLBI**: TLB Invalidate operation: an alias of SYS.

**TSB CSYNC**: Trace Synchronization Barrier.

**TST (immediate)**: Test bits (immediate): an alias of ANDS (immediate).

**TST (shifted register)**: Test (shifted register): an alias of ANDS (shifted register).

**UBFIZ**: Unsigned Bitfield Insert in Zero: an alias of UBFM.

**UBFM**: Unsigned Bitfield Move.

**UBFX**: Unsigned Bitfield Extract: an alias of UBFM.

**UFD**: Permanently Undefined.

**UDIV**: Unsigned Divide.

**UMADDL**: Unsigned Multiply-Add Long.

**UMNEGL**: Unsigned Multiply-Negate Long: an alias of UMSUBL.

**UMSUBL**: Unsigned Multiply-Subtract Long.

**UMULH**: Unsigned Multiply High.

**UMULL**: Unsigned Multiply Long: an alias of UMADDL.

**UXTB**: Unsigned Extend Byte: an alias of UBFM.

**UXTH**: Unsigned Extend Halfword: an alias of UBFM.

**WFE**: Wait For Event.

**WFI**: Wait For Interrupt.

**XAFlag**: Convert floating-point condition flags from external format to ARM format.

**XPACD, XPACI, XPACLRI**: Strip Pointer Authentication Code.

**YIELD**: YIELD.
ABS: Absolute value (vector).
ADD (vector): Add (vector).
ADDHN, ADDHN2: Add returning High Narrow.
ADDP (scalar): Add Pair of elements (scalar).
ADDP (vector): Add Pairwise (vector).
ADDV: Add across Vector.
AESD: AES single round decryption.
AESE: AES single round encryption.
AESIMC: AES inverse mix columns.
AESMC: AES mix columns.
AND (vector): Bitwise AND (vector).
BCAX: Bit Clear and XOR.
BIC (vector, immediate): Bitwise bit Clear (vector, immediate).
BIF: Bitwise Insert if False.
BIT: Bitwise Insert if True.
BSL: Bitwise Select.
CLS (vector): Count Leading Sign bits (vector).
CLZ (vector): Count Leading Zero bits (vector).
CMEQ (register): Compare bitwise Equal (vector).
CMEQ (zero): Compare bitwise Equal to zero (vector).
CMGE (register): Compare signed Greater than or Equal (vector).
CMGE (zero): Compare signed Greater than or Equal to zero (vector).
CMGT (register): Compare signed Greater than (vector).
CMGT (zero): Compare signed Greater than zero (vector).
CMHI (register): Compare unsigned Higher (vector).
CMHS (register): Compare unsigned Higher or Same (vector).
CMLE (zero): Compare signed Less than or Equal to zero (vector).
CMLT (zero): Compare signed Less than zero (vector).
CMTST: Compare bitwise Test bits nonzero (vector).
CNT: Population Count per byte.
DUP (element): Duplicate vector element to vector or scalar.
DUP (general): Duplicate general-purpose register to vector.
EOR (vector): Bitwise Exclusive OR (vector).
EOR3: Three-way Exclusive OR.

EXT: Extract vector from pair of vectors.

FABD: Floating-point Absolute Difference (vector).

FABS (scalar): Floating-point Absolute value (scalar).

FABS (vector): Floating-point Absolute value (vector).

FACGE: Floating-point Absolute Compare Greater than or Equal (vector).

FACGT: Floating-point Absolute Compare Greater than (vector).

FADD (scalar): Floating-point Add (scalar).

FADD (vector): Floating-point Add (vector).

FADDP (scalar): Floating-point Add Pair of elements (scalar).

FADDP (vector): Floating-point Add Pairwise (vector).

FCADD: Floating-point Complex Add.

FCCMP: Floating-point Conditional quiet Compare (scalar).

FCCMPE: Floating-point Conditional signaling Compare (scalar).

FCMEQ (register): Floating-point Compare Equal (vector).

FCMEQ (zero): Floating-point Compare Equal to zero (vector).

FCMGE (register): Floating-point Compare Greater than or Equal (vector).

FCMGE (zero): Floating-point Compare Greater than or Equal to zero (vector).

FCMG (register): Floating-point Compare Greater than (vector).

FCMG (zero): Floating-point Compare Greater than zero (vector).

FCMLA: Floating-point Complex Multiply Accumulate.

FCMLA (by element): Floating-point Complex Multiply Accumulate (by element).

FCMLE (zero): Floating-point Compare Less than or Equal to zero (vector).

FCMLT (zero): Floating-point Compare Less than zero (vector).

FCMP: Floating-point quiet Compare (scalar).

FCMP: Floating-point signaling Compare (scalar).

FCSEL: Floating-point Conditional Select (scalar).

FCVT: Floating-point Convert precision (scalar).

FCVTAS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).

FCVTAS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).

FCVT (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).

FCVT (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).

FCVT (scalar): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).

FCVT (vector): Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).

FCVTMU (scalar): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).
FCVTMU (vector): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).
FCVTN, FCVTN2: Floating-point Convert to lower precision Narrow (vector).
FCVTNS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar).
FCVTNS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).
FCVTN (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).
FCVTN (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).
FCVTNS (scalar): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).
FCVTNS (vector): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).
FCVTNU (scalar): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).
FCVTNU (vector): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).
FCVTXN, FCVTXN2: Floating-point Convert to lower precision Narrow, rounding to odd (vector).
FCVTS (scalar, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).
FCVTS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).
FCVTS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).
FCVTZU (scalar, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).
FCVTZU (vector, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
FCVTZU (scalar, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).
FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).
FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).
FJCVTZS: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.
FMADD: Floating-point fused Multiply-Add (scalar).
FMAX (scalar): Floating-point Maximum (scalar).
FMAX (vector): Floating-point Maximum (vector).
FMAXNM (scalar): Floating-point Maximum Number (scalar).
FMAXNM (vector): Floating-point Maximum Number (vector).
FMAXNMP (scalar): Floating-point Maximum Number of Pair of elements (scalar).
FMAXNMP (vector): Floating-point Maximum Number Pairwise (vector).
FMAXMV: Floating-point Maximum Number across Vector.
FMAXP (scalar): Floating-point Maximum of Pair of elements (scalar).
FMAXP (vector): Floating-point Maximum Pairwise (vector).
FMAXV: Floating-point Maximum across Vector.
FMIN (scalar): Floating-point Minimum (scalar).
FMIN (vector): Floating-point minimum (vector).
FMINNM (scalar): Floating-point Minimum Number (scalar).
FMINNM (vector): Floating-point Minimum Number (vector).
FMINNM (scalar): Floating-point Minimum Number of Pair of elements (scalar).
FMINNM (vector): Floating-point Minimum Number Pairwise (vector).
FMINMV: Floating-point Minimum Number across Vector.
FMINP (scalar): Floating-point Minimum of Pair of elements (scalar).
FMINP (vector): Floating-point Minimum Pairwise (vector).
FMINV: Floating-point Minimum across Vector.
FMLA (by element): Floating-point fused Multiply-Add to accumulator (by element).
FMLA (vector): Floating-point fused Multiply-Add to accumulator (vector).
FMLAL, FMLAL2 (by element): Floating-point fused Multiply-Add Long to accumulator (by element).
FMLAL, FMLAL2 (vector): Floating-point fused Multiply-Add Long to accumulator (vector).
FMLS (by element): Floating-point fused Multiply-Subtract from accumulator (by element).
FMLS (vector): Floating-point fused Multiply-Subtract from accumulator (vector).
FMLSL, FMLSL2 (by element): Floating-point fused Multiply-Subtract Long from accumulator (by element).
FMLSL, FMLSL2 (vector): Floating-point fused Multiply-Subtract Long from accumulator (vector).
FMOV (general): Floating-point Move to or from general-purpose register without conversion.
FMOV (register): Floating-point Move register without conversion.
FMOV (scalar, immediate): Floating-point move immediate (scalar).
FMOV (vector, immediate): Floating-point move immediate (vector).
FMSUB: Floating-point Fused Multiply-Subtract (scalar).
FMUL (by element): Floating-point Multiply (by element).
FMUL (scalar): Floating-point Multiply (scalar).
FMUL (vector): Floating-point Multiply (vector).
FMULX: Floating-point Multiply extended.
FMULX (by element): Floating-point Multiply extended (by element).
FNEG (scalar): Floating-point Negate (scalar).
FNEG (vector): Floating-point Negate (vector).
FNMADD: Floating-point Negated fused Multiply-Add (scalar).
FNMSUB: Floating-point Negated fused Multiply-Subtract (scalar).
FNMUL (scalar): Floating-point Multiply-Negate (scalar).
FRECPE: Floating-point Reciprocal Estimate.
FRECPS: Floating-point Reciprocal Step.
FRECPS: Floating-point Reciprocal exponent (scalar).
PRINT32X (scalar): Floating-point Round to 32-bit Integer, using current rounding mode (scalar).
PRINT32X (vector): Floating-point Round to 32-bit Integer, using current rounding mode (vector).
PRINT32Z (scalar): Floating-point Round to 32-bit Integer toward Zero (scalar).
FRINT32Z (vector): Floating-point Round to 32-bit Integer toward Zero (vector).
FRINT64X (scalar): Floating-point Round to 64-bit Integer, using current rounding mode (scalar).
FRINT64X (vector): Floating-point Round to 64-bit Integer, using current rounding mode (vector).
FRINT64Z (scalar): Floating-point Round to 64-bit Integer toward Zero (scalar).
FRINT64Z (vector): Floating-point Round to 64-bit Integer toward Zero (vector).
FRINTA (scalar): Floating-point Round to Integral, to nearest with ties to Away (scalar).
FRINTA (vector): Floating-point Round to Integral, to nearest with ties to Away (vector).
FRINTI (scalar): Floating-point Round to Integral, using current rounding mode (scalar).
FRINTI (vector): Floating-point Round to Integral, using current rounding mode (vector).
FRINTM (scalar): Floating-point Round to Integral, toward Minus infinity (scalar).
FRINTM (vector): Floating-point Round to Integral, toward Minus infinity (vector).
FRINTN (scalar): Floating-point Round to Integral, to nearest with ties to even (scalar).
FRINTN (vector): Floating-point Round to Integral, to nearest with ties to even (vector).
FRINTP (scalar): Floating-point Round to Integral, toward Plus infinity (scalar).
FRINTP (vector): Floating-point Round to Integral, toward Plus infinity (vector).
FRINTX (scalar): Floating-point Round to Integral exact, using current rounding mode (scalar).
FRINTX (vector): Floating-point Round to Integral exact, using current rounding mode (vector).
FRINTZ (scalar): Floating-point Round to Integral, toward Zero (scalar).
FRINTZ (vector): Floating-point Round to Integral, toward Zero (vector).
FRSQRTE: Floating-point Reciprocal Square Root Estimate.
FRSQRTS: Floating-point Reciprocal Square Root Step.
FSQRT (scalar): Floating-point Square Root (scalar).
FSQRT (vector): Floating-point Square Root (vector).
FSUB (scalar): Floating-point Subtract (scalar).
FSUB (vector): Floating-point Subtract (vector).
INS (element): Insert vector element from another vector element.
INS (general): Insert vector element from general-purpose register.
LD1 (multiple structures): Load multiple single-element structures to one, two, three, or four registers.
LD1 (single structure): Load one single-element structure to one lane of one register.
LD2 (multiple structures): Load multiple 2-element structures to two registers.
LD2 (single structure): Load single 2-element structure to one lane of two registers.
LD3 (multiple structures): Load multiple 3-element structures to three registers.
LD3 (single structure): Load single 3-element structure to one lane of three registers.
LD3R: Load single 3-element structure and Replicate to all lanes of three registers.
LD4 (multiple structures): Load multiple 4-element structures to four registers.

LD4 (single structure): Load single 4-element structure to one lane of four registers.

LD4R: Load single 4-element structure and Replicate to all lanes of four registers.

LDNP (SIMD&FP): Load Pair of SIMD&FP registers, with Non-temporal hint.

LDP (SIMD&FP): Load Pair of SIMD&FP registers.

LDR (immediate, SIMD&FP): Load SIMD&FP Register (immediate offset).

LDR (literal, SIMD&FP): Load SIMD&FP Register (PC-relative literal).

LDR (register, SIMD&FP): Load SIMD&FP Register (register offset).

LDUR (SIMD&FP): Load SIMD&FP Register (unscaled offset).

MLA (by element): Multiply-Add to accumulator (vector, by element).

MLA (vector): Multiply-Add to accumulator (vector).

MLS (by element): Multiply-Subtract from accumulator (vector, by element).

MLS (vector): Multiply-Subtract from accumulator (vector).

MOV (element): Move vector element to another vector element: an alias of INS (element).

MOV (from general): Move general-purpose register to a vector element: an alias of INS (general).

MOV (scalar): Move vector element to scalar: an alias of DUP (element).

MOV (to general): Move vector element to general-purpose register: an alias of UMOV.


MOV: Move Immediate (vector).

MUL (by element): Multiply (vector, by element).

MUL (vector): Multiply (vector).

MVN: Bitwise NOT (vector): an alias of NOT.

MVNI: Move inverted Immediate (vector).

NEG (vector): Negate (vector).

NOT: Bitwise NOT (vector).

ORN (vector): Bitwise inclusive OR NOT (vector).

ORR (vector, immediate): Bitwise inclusive OR (vector, immediate).

ORR (vector, register): Bitwise inclusive OR (vector, register).

PMUL: Polynomial Multiply.

PMULL, PMULL2: Polynomial Multiply Long.

RADDHN, RADDHN2: Rounding Add returning High Narrow.

RAX1: Rotate and Exclusive OR.

RBIT (vector): Reverse Bit order (vector).

REV16 (vector): Reverse elements in 16-bit halfwords (vector).

REV32 (vector): Reverse elements in 32-bit words (vector).

REV64: Reverse elements in 64-bit doublewords (vector).
RSHRN, RSHRN2: Rounding Shift Right Narrow (immediate).
RSUBHN, RSUBHN2: Rounding Subtract returning High Narrow.
SABA: Signed Absolute difference and Accumulate.
SABAL, SABAL2: Signed Absolute difference and Accumulate Long.
SABD: Signed Absolute Difference.
SABDL, SABDL2: Signed Absolute Difference Long.
SADALP: Signed Add and Accumulate Long Pairwise.
SADDL, SADDL2: Signed Add Long (vector).
SADLPL: Signed Add Long Pairwise.
SADDLV: Signed Add Long across Vector.
SADDW, SADDW2: Signed Add Wide.
SCVTF (scalar, fixed-point): Signed fixed-point Convert to Floating-point (scalar).
SCVTF (scalar, integer): Signed integer Convert to Floating-point (scalar).
SCVTF (vector, fixed-point): Signed fixed-point Convert to Floating-point (vector).
SCVTF (vector, integer): Signed integer Convert to Floating-point (vector).
SDOT (by element): Dot Product signed arithmetic (vector, by element).
SDOT (vector): Dot Product signed arithmetic (vector).
SHA1C: SHA1 hash update (choose).
SHA1H: SHA1 fixed rotate.
SHA1M: SHA1 hash update (majority).
SHA1P: SHA1 hash update (parity).
SHA1SU0: SHA1 schedule update 0.
SHA1SU1: SHA1 schedule update 1.
SHA256H: SHA256 hash update (part 1).
SHA256H2: SHA256 hash update (part 2).
SHA256SU0: SHA256 schedule update 0.
SHA256SU1: SHA256 schedule update 1.
SHA512H: SHA512 Hash update part 1.
SHA512H2: SHA512 Hash update part 2.
SHA512SU0: SHA512 Schedule Update 0.
SHA512SU1: SHA512 Schedule Update 1.
SHADD: Signed Halving Add.
SHL: Shift Left (immediate).
SHLL, SHLL2: Shift Left Long (by element size).
SHRN, SHRN2: Shift Right Narrow (immediate).
SHSUB: Signed Halving Subtract.
SLI: Shift Left and Insert (immediate).
SM3PARTW1: SM3PARTW1.
SM3PARTW2: SM3PARTW2.
SM3S1: SM3S1.
SM3T1A: SM3T1A.
SM3T1B: SM3T1B.
SM3T2A: SM3T2A.
SM3T2B: SM3T2B.
SM4E: SM4 Encode.
SM4EKEY: SM4 Key.
SMAX: Signed Maximum (vector).
SMAVP: Signed Maximum Pairwise.
SMAVY: Signed Maximum across Vector.
SMIN: Signed Minimum (vector).
SMINP: Signed Minimum Pairwise.
SMINV: Signed Minimum across Vector.
SMLSL, SMLSL2 (by element): Signed Multiply-Subtract Long (vector, by element).
SMOV: Signed Move vector element to general-purpose register.
SQABS: Signed saturating Absolute value.
SQADD: Signed saturating Add.
SQDMLSL, SQDMLSL2 (by element): Signed saturating Doubling Multiply-Subtract Long (by element).
SQDMULH (by element): Signed saturating Doubling Multiply returning High half (by element).
SQDMULH (vector): Signed saturating Doubling Multiply returning High half.
SQDMDLL, SQDMDLL2 (by element): Signed saturating Doubling Multiply Long (by element).
SQNEG: Signed saturating Negate.
SQRDMLAH (by element): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).
SQRDMLAH (vector): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).
SQRDMLSH (by element): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

SQRDMLSH (vector): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).

SQRDMULH (by element): Signed saturating Rounding Doubling Multiply returning High half (by element).

SQRDMULH (vector): Signed saturating Rounding Doubling Multiply returning High half.

SQRSHL: Signed saturating Rounding Shift Left (register).

SQRSHRN, SQRSHRN2: Signed saturating Rounded Shift Right Narrow (immediate).

SQRSHRUN, SQRSHRUN2: Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

SQSHL (immediate): Signed saturating Shift Left (immediate).

SQSHL (register): Signed saturating Shift Left (register).

SQSHLU: Signed saturating Shift Left Unsigned (immediate).

SQSHRN, SQSHRN2: Signed saturating Shift Right Narrow (immediate).

SQSHRUN, SQSHRUN2: Signed saturating Shift Right Unsigned Narrow (immediate).

SQSUB: Signed saturating Subtract.

SOXTN, SOXTN2: Signed saturating extract Narrow.

SOXTUN, SOXTUN2: Signed saturating extract Unsigned Narrow.

SRHADD: Signed Rounding Halving Add.

SRI: Shift Right and Insert (immediate).

SRSHL: Signed Rounding Shift Left (register).

SRSHR: Signed Rounding Shift Right (immediate).

SRSRA: Signed Rounding Shift Right and Accumulate (immediate).

SSHLL, SSHLL2: Signed Shift Left Long (immediate).

SSHR: Signed Shift Right (immediate).

SSRA: Signed Shift Right and Accumulate (immediate).

SSUBL, SSUBL2: Signed Subtract Long.

SSUBW, SSUBW2: Signed Subtract Wide.

ST1 (multiple structures): Store multiple single-element structures from one, two, three, or four registers.

ST1 (single structure): Store a single-element structure from one lane of one register.

ST2 (multiple structures): Store multiple 2-element structures from two registers.

ST2 (single structure): Store single 2-element structure from one lane of two registers.

ST3 (multiple structures): Store multiple 3-element structures from three registers.

ST3 (single structure): Store single 3-element structure from one lane of three registers.

ST4 (multiple structures): Store multiple 4-element structures from four registers.

ST4 (single structure): Store single 4-element structure from one lane of four registers.

STNP (SIMD&FP): Store Pair of SIMD&FP registers, with Non-temporal hint.

STP (SIMD&FP): Store Pair of SIMD&FP registers.
STR (immediate, SIMD&FP): Store SIMD&FP register (immediate offset).

STR (register, SIMD&FP): Store SIMD&FP register (register offset).

STUR (SIMD&FP): Store SIMD&FP register (unscaled offset).

SUB (vector): Subtract (vector).

SUBHN, SUBHN2: Subtract returning High Narrow.

SUQADD: Signed saturating Accumulate of Unsigned value.


TBL: Table vector Lookup.

TBX: Table vector lookup extension.

TRN1: Transpose vectors (primary).

TRN2: Transpose vectors (secondary).

UABA: Unsigned Absolute difference and Accumulate.

UABAL, UABAL2: Unsigned Absolute difference and Accumulate Long.

UABD: Unsigned Absolute Difference (vector).

UABDL, UABDL2: Unsigned Absolute Difference Long.

UADALP: Unsigned Add and Accumulate Long Pairwise.

UADDL, UADDL2: Unsigned Add Long (vector).

UADDLP: Unsigned Add Long Pairwise.

UADDLV: Unsigned sum Long across Vector.

UADDW, UADDW2: Unsigned Add Wide.

UCVTF (scalar, fixed-point): Unsigned fixed-point Convert to Floating-point (scalar).

UCVTF (scalar, integer): Unsigned integer Convert to Floating-point (scalar).

UCVTF (vector, fixed-point): Unsigned fixed-point Convert to Floating-point (vector).

UCVTF (vector, integer): Unsigned integer Convert to Floating-point (vector).

UDOT (by element): Dot Product unsigned arithmetic (vector, by element).

UDOT (vector): Dot Product unsigned arithmetic (vector).

UHADD: Unsigned Halving Add.

UHSUB: Unsigned Halving Subtract.

UMAX: Unsigned Maximum (vector).

UMAXP: Unsigned Maximum Pairwise.

UMAXV: Unsigned Maximum across Vector.

UMIN: Unsigned Minimum (vector).

UMINP: Unsigned Minimum Pairwise.

UMINV: Unsigned Minimum across Vector.


**UMLSL, UMLSL2 (by element)**: Unsigned Multiply-Subtract Long (vector, by element).

**UMLSL, UMLSL2 (vector)**: Unsigned Multiply-Subtract Long (vector).

**UMOV**: Unsigned Move vector element to general-purpose register.

**UMULL, UMULL2 (by element)**: Unsigned Multiply Long (vector, by element).

**UMULL, UMULL2 (vector)**: Unsigned Multiply long (vector).

**UADD**: Unsigned saturating Add.

**UQRSHL**: Unsigned saturating Rounding Shift Left (register).

**UQRSHRN, UQRSHRN2**: Unsigned saturating Rounded Shift Right Narrow (immediate).

**UQADD (immediate)**: Unsigned saturating Shift Left (immediate).

**UQADD (register)**: Unsigned saturating Shift Left (register).

**UQSHRN, UQSHRN2**: Unsigned saturating Shift Right Narrow (immediate).

**UQSHL**: Unsigned saturating Shift Left (immediate).

**UQSHL (register)**: Unsigned saturating Shift Left (register).

**UQSUB**: Unsigned saturating Subtract.

**UQXTN, UQXTN2**: Unsigned saturating extract Narrow.

**URECPE**: Unsigned Reciprocal Estimate.

**URHADD**: Unsigned Rounding Halving Add.

**URSHL**: Unsigned Rounding Shift Left (register).

**URSHR**: Unsigned Rounding Shift Right (immediate).

**URSQRT**: Unsigned Reciprocal Square Root Estimate.

**URSUB**: Unsigned Rounding Shift Right and Accumulate (immediate).

**USHL**: Unsigned Shift Left (register).

**USHLL, USHLL2**: Unsigned Shift Left Long (immediate).

**USHR**: Unsigned Shift Right (immediate).

**USQADD**: Unsigned saturating Accumulate of Signed value.

**USRA**: Unsigned Shift Right and Accumulate (immediate).

**USUBL, USUBL2**: Unsigned Subtract Long.

**USUBW, USUBW2**: Unsigned Subtract Wide.

**UXTL, UXTL2**: Unsigned extend Long: an alias of USHLL, USHLL2.

**UZP1**: Unzip vectors (primary).

**UZP2**: Unzip vectors (secondary).

**XAR**: Exclusive OR and Rotate.

**XTN, XTN2**: Extract Narrow.

**ZIP1**: Zip vectors (primary).

**ZIP2**: Zip vectors (secondary).
ABS

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |    |    |    |    |    |    |    |    |    |    |

U

Scalar

ABS <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0  | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |    |    |    |    |    |    |    |    |    |    |

U

Vector

ABS <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”: 

ABS
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADC

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.

```
<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

32-bit (sf == 0)

ADC <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

ADC <Xd>, <Xn>, <Xm>

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;
if sub_op then
    operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADCS

Add with Carry, setting flags, adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ADCS <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

ADCS <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;

if sub_op then
    operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

sf 0 0 0 1 0 1 1 0 0 1 Rm option imm3 Rn Rd

op S

32-bit (sf == 0)

ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

64-bit (sf == 1)

ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;

Assembler Symbols

<Wd|WSP>  Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP>  Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm>  Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

Xd|SP>  Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP>  Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R>  Is a width specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

<m>  Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

ADD (extended register)
If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```plaintext
bits(datasize) result;
bites(datasize) operand1 = if n == 31 then SP[ ] else X[n];
bites(datasize) operand2 = ExtendReg(m, extend_type, shift);
bites(4) nzcv;
bit carry_in;
if sub_op then
    operand2 = NOT(operand2);
carry_in = '1';
else
    carry_in = '0';
(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
if d == 31 && !setflags then
    SP[ ] = result;
else
    X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (immediate)

Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (to/from SP).

```
<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>!= 1x</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

32-bit (sf == 0)

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

```plaintext
type integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
  when '10' SEE "ADDG, SUBG";
  when '11' ReservedValue();
```

Assembler Symbols

- `<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "shift<0>":

<table>
<thead>
<tr>
<th>shift&lt;0&gt;</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (to/from SP)</td>
<td>shift == '00' &amp;&amp; imm12 == '000000000000' &amp;&amp; (Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

```plaintext
bits(datosize) result;
bits(datosize) operand1 = if n == 31 then SP[] else X[n];
bits(datosize) operand2 = imm;
bits(4) nzcv;
bit carry_in;
if sub_op then
  operand2 = NOT(operand2);
  carry_in = '1';
else
  carry_in = '0';
(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
if setflags then
  PSTATE.<N,Z,C,V> = nzcv;
if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

32-bit (sf == 0)

ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
**Operation**

```c
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (vector)

Add (vector). This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Vector**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler Symbols**

[V] Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

d Is the number of the SIMD&FP destination register, in the "Rd" field.

n Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

m Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

[Vd] Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDG

Add with Tag adds an immediate value scaled by the Tag granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

Integer

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | uimm6 | (0) | (0) | uimm4 | Xn  | Xd  |

ADDG (<Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>)

integer d = UInt(Xd);
integer n = UInt(Xn);
bits(4) tag_offset = uimm4;
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
boolean ADD = TRUE;

Assembler Symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
<uimm6> Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
<uimm4> Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

Operation

bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;
if AllocationTagAccessIsEnabled() then
  rtag = ChooseNonExcludedTag(start_tag, tag_offset, exclude);
else
  rtag = '0000';
if ADD then
  (result, -) = AddWithCarry(operand1, offset, '0');
else
  (result, -) = AddWithCarry(operand1, NOT(offset), '1');
result = AddressWithAllocationTag(result, rtag);
if d == 31 then
  SP[] = result;
else
  X[d] = result;
ADDHN, ADDHN2

Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are truncated. For rounded results, see RADDHN.

The ADDHN instruction writes the vector to the lower half of the destination register and clears the upper half; while the ADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

ADDHN[2] <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize)   result;
int round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;
for e = 0 to elements-1
   element1 = Elem[operand1, e, 2*esize];
   element2 = Elem[operand2, e, 2*esize];
   if sub_op then
      sum = element1 - element2;
   else
      sum = element1 + element2;
   sum = sum + round_const;
   Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDP (scalar)

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Register Table]

Advanced SIMD

ADDP <V><d>, <Vn>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;

integer esize = 8 << UInt(size);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = ReduceOp_ADD;
```

Assembler Symbols

- **<V>** Is the destination width specifier, encoded in “size”:
  - size
    - 0x: RESERVED
    - 10: RESERVED
    - 11: 2D

- **<d>** Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- **<T>** Is the source arrangement specifier, encoded in “size”:
  - size
    - 0x: RESERVED
    - 10: RESERVED
    - 11: 2D

Operation

```plaintext
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDP (vector)

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Three registers of the same type

$ADDP <Vd>..<T>, <Vn>..<T>, <Vm>..<T>$

```plaintext
integer d = UInt (Rd);
integer n = UInt (Rn);
integer m = UInt (Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assemblyer Symbols

- **<Vd>**
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<T>**
  - Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- **<Vn>**
  - Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- **<Vm>**
  - Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
**ADDS (extended register)**

Add (extended register), setting flags, adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the \( \langle Rm \rangle \) register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias \( \text{CMN (extended register)} \).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| sf | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | Rm  | option  | imm3  | Rn  | Rd  |

**32-bit (sf == 0)**

\[
\text{ADDS} <Wd>, <Wn|WSP>, <Wm>{, <extend> \{#<amount>\}}
\]

**64-bit (sf == 1)**

\[
\text{ADDS} <Xd>, <Xn|SP>, <R><m>{, <extend> \{#<amount>\}}
\]

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
integer datasize = if \(sf == '1'\) then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;

**Assembler Symbols**

- \( <Wd> \): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( <Wn|WSP> \): Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- \( <Wm> \): Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \( <Xd> \): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( <Xn|SP> \): Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- \( <R> \): Is a width specifier, encoded in "option":
  - option  | \(<R>\) |
  | 00x  | W  |
  | 010  | W  |
  | x11  | X  |
  | 10x  | W  |
  | 110  | W  |
- \( <m> \): Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- \( <\text{extend}> \): For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":
  - option  | \(<\text{extend}>\) |
  | 000  | UXTB |
  | 001  | UXTH |
  | 010  | LSL|UXTW |
  | 011  | UXTX |
  | 100  | SXTB |
  | 101  | SXTX |
  | 110  | SXTW |
  | 111  | SXTX |

If "Rn" is '111111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases "<extend>" is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

---

Page 43
If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (extended register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;
if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDS (immediate)

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **CMN (immediate)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>!= 1x</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ADDS <Wd>, <Wn|WSP>, #<imm>{, <shift>}

64-bit (sf == 1)

ADDS <Xd>, <Xn|SP>, #<imm>{, <shift>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;
case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
  when '10' SEE "ADDG, SUBG";
  when '11' ReservedValue();

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

<table>
<thead>
<tr>
<th>shift&lt;0&gt;</th>
<th>shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (immediate)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDS (shifted register)

Add (shifted register), setting flags, adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **CMN (shifted register)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ADDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

ADDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Xm>** Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<shift>** Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<amount>** For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

```c
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;
if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ADDV**

Add across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 1   | 1   | 0   | 1   | 1   | 0   | 0   | 1   | 1   | 1   | 0   | Rd  |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
```

**Advanced SIMD**

```
ADDV <V><d>, <Vn>.<T>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = ReduceOp_ADD;
```

**Assembler Symbols**

- `<V>`: Is the destination width specifier, encoded in “size”:
  ```
  size  <V>
  00    B
  01    H
  10    S
  11    RESERVED
  ```

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<T>`: Is an arrangement specifier, encoded in “size:Q”:
  ```
  size  Q  <T>
  00    0    8B
  00    1    16B
  01    0    4H
  01    1    8H
  10    0    RESERVED
  10    1    4S
  11    x    RESERVED
  ```

**Operation**

```
CheckFPAAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| op | immlo| 1  | 0  | 0  | 0  | immhi| Rd |

Literal

ADR \(<Xd>, <label>\)

integer \(d\) = \(\text{UInt}(\text{Rd})\);
boolean \(\text{page}\) = \(\{\text{op} = 1\}\);
bits(64) \(\text{imm}\);

if \(\text{page}\) then
  \(\text{imm} = \text{SignExtend}(\text{immhi}:\text{immlo}:\text{Zeros}(12), 64)\);
else
  \(\text{imm} = \text{SignExtend}(\text{immhi}:\text{immlo}, 64)\);

Assembler Symbols

\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<label>\) Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

Operation

bits(64) \(\text{base} = \text{PC}[]\);

if \(\text{page}\) then
  \(\text{base}<>11:0> = \text{Zeros}(12)\);

\(X[d] = \text{base} + \text{imm}\);
ADRP

Form PC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | immlo | 0  | 0  | 0  | 0  | immhi |               | Rd |

op

**Literal**

```
ADRP <Xd>, <label>

integer d = UInt(Rd);
boolean page = (op == '1');
bits(64) imm;

if page then
    imm = SignExtend(immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(immhi:immlo, 64);
```

**Assembler Symbols**

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<label>` Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096.

**Operation**

```
bits(64) base = PC[];

if page then
    base<11:0> = Zeros(12);

Xd[d] = base + imm;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AESD

AES single round decryption.

Advanced SIMD

AESD \(<Vd>\), \(<Vn>\)

**Assembler Symbols**

\(<Vd>\) Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

**Operation**

\[
\text{AArch64.CheckFPAdvSIMDEnabled}();
\]

\[
\begin{align*}
\text{bits}(128) \ &\text{operand1} = \ V[d]; \\
\text{bits}(128) \ &\text{operand2} = \ V[n]; \\
\text{bits}(128) \ &\text{result}; \\
\text{result} = \text{operand1 EOR operand2}; \\
\text{if decrypt then} \\
&\quad \text{result} = \text{AESInvSubBytes(AESInvShiftRows(result))}; \\
\text{else} \\
&\quad \text{result} = \text{AESSubBytes(AESShiftRows(result))}; \\
\V[d] = \text{result};
\end{align*}
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AES single round encryption.

Advanced SIMD

AESE <Vd>.16B, <Vn>.16B

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESEExt() then UNDEFINED;
boolean decrypt = (D == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();
bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
if decrypt then
    result = AESInvSubBytes(AESInvShiftRows(result));
else
    result = AESSubBytes(AESShiftRows(result));
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESIMC

AES inverse mix columns.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rn | Rd |

Advanced SIMD

AESIMC <Vd>.16B, <Vn>.16B

integer d = Uint(Rd);
integer n = Uint(Rn);
if !HaveAESExt() then UNDEFINED;
boolean decrypt = (D == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();
bits(128) operand = V[n];
bits(128) result;
if decrypt then
result = AESInvMixColumns(operand);
else
result = AESMixColumns(operand);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AESMC

AES mix columns.

Advanced SIMD

AESMC <Vd>.16B, <Vn>.16B

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
boolean decrypt = (D == '1');
```

Assembler Symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```plaintext
AArch64.CheckFPAdvSIMDEnabled();
bits(128) operand = V[n];
bits(128) result;
if decrypt then
    result = AESInvMixColumns(operand);
else
    result = AESMixColumns(operand);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register. Depending on the settings in the \( CPACR_{EL1} \), \( CPTR_{EL2} \), and \( CPTR_{EL3} \) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

\[
\begin{array}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
AND (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 sf 0 0 1 0 0 1 0 0 N immr | imms | Rn | Rd
 opc

32-bit (sf == 0 & N == 0)

AND <Wd|WSP>, <Wn>, #<imm>

64-bit (sf == 1)

AND <Xd|SP>, <Xn>, #<imm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;
case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;
if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
AND (shifted register)

Bitwise AND (shifted register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

AND <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

AND <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
Operation

```c
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ANVS (immediate)

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **TST (immediate)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**32-bit (sf == 0 & N == 0)**

ANVS <Wd>, <Wn>, #<imm>

**64-bit (sf == 1)**

ANVS <Xd>, <Xn>, #<imm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<imm>**
  - For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  - For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST (immediate)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

```c
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ANDS (shifted register)**

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **TST (shifted register)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

```
ANDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
ANDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of ASRV. This means:

- The encodings in this description are named to match the encodings of ASRV.
- The description of ASRV gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |

32-bit (sf == 0)

ASR <Wd>, <Wn>, <Wm>

is equivalent to

ASRV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

ASR <Xd>, <Xn>, <Xm>

is equivalent to

ASRV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of ASRV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 0 1 0 1 0 N</th>
<th>immr</th>
<th>x 1 1 1 1 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>
| opc | imms

32-bit (sf == 0 & N == 0 & imms == 011111)

ASR <Wd>, <Wn>, #<shift>

is equivalent to

SBFM <Wd>, <Wn>, #<shift>, #31

and is always the preferred disassembly.

64-bit (sf == 1 & N == 1 & imms == 11111)

ASR <Xd>, <Xn>, #<shift>

is equivalent to

SBFM <Xd>, <Xn>, #<shift>, #63

and is always the preferred disassembly.

Assembler Symbols

<Wd>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<brk><Wn>  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<brk><Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<brk><Xn>  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<brk><shift>  For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.
<brk>For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ASRV

Arithmetic Shift Right Variable shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias \texttt{ASR (register)}.

<table>
<thead>
<tr>
<th>sf</th>
<th>Rm</th>
<th>op2</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

\texttt{ASRV <Wd>, <Wn>, <Wm>}

64-bit (sf == 1)

\texttt{ASRV <Xd>, <Xn>, <Xm>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

ShiftType shift_type = DecodeShift(op2);
```

\textbf{Assembler Symbols}

\texttt{<Wd>} Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\texttt{<Wn>} Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

\texttt{<Wm>} Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.

\texttt{<Xd>} Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\texttt{<Xn>} Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

\texttt{<Xm>} Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

\textbf{Operation}

```
bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

\textbf{Operational information}

If \texttt{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Address Translate. For more information, see .

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | op1 | 0  | 1  | 1  | 1  | 0  | 0  | x  | op2 | Rt |

System

AT <at_op>, <Xt>

is equivalent to

SYS #<op1>, C7, <Cm>, #<op2>, <Xt>

and is the preferred disassembly when SysOp(op1,'0111',CRm,op2) == Sys_AT.

Assembler Symbols

<at_op> Is an AT instruction name, as listed for the AT system instruction group, encoded in “op1:CRm<0>:op2”:

<table>
<thead>
<tr>
<th>op1</th>
<th>CRm&lt;0&gt;</th>
<th>op2</th>
<th>&lt;at_op&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>000</td>
<td>S1E1R</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>001</td>
<td>S1E1W</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>010</td>
<td>S1E0R</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>011</td>
<td>S1E0W</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>000</td>
<td>S1E1RP</td>
<td>ARMv8.2-ATS1E1</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>001</td>
<td>S1E1WP</td>
<td>ARMv8.2-ATS1E1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>000</td>
<td>S1E2R</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>001</td>
<td>S1E2W</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>100</td>
<td>S12E1R</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>101</td>
<td>S12E1W</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>110</td>
<td>S12E0R</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>111</td>
<td>S12E0W</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>000</td>
<td>S1E3R</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>001</td>
<td>S1E3W</td>
<td>-</td>
</tr>
</tbody>
</table>

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.
AUTDA, AUTDZA

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A. The address is in the general-purpose register that is specified by <Xd>.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTDA.
- The value zero, for AUTDZA.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Integer

\[
\begin{array}{cccccccccccccccccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & Z & 1 & 1 & 0 & & Rn & & Rd
\end{array}
\]

AUTDA (Z == 0)

AUTDA <Xd>, <Xn|SP>

AUTDZA (Z == 1 && Rn == 1111)

AUTDZA <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // AUTDA
    if n == 31 then source_is_sp = TRUE;
else // AUTDZA
    if n != 31 then UNDEFINED;
```

Assembler Symbols

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```java
if source_is_sp then
    X[d] = AuthDA(X[d], SP());
else
    X[d] = AuthDA(X[d], X[n]);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00be8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**AUTDB, AUTDZB**

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B. The address is in the general-purpose register that is specified by \(<Xd>\). The modifier is:

- In the general-purpose register or stack pointer that is specified by \(<Xn|SP>\) for AUTDB.
- The value zero, for AUTDZB.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

### Integer (ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 1  | 1  | 1  | Rn | | Rd |

#### AUTDB (Z == 0)

\[
\text{AUTDB} \ <Xd>, \ <Xn|SP>
\]

#### AUTDZB (Z == 1 \&\& Rn == 1111)

\[
\text{AUTDZB} \ <Xd>
\]

```plaintext
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
   UNDEFINED;
if Z == '0' then // AUTDB
   if n == 31 then source_is_sp = TRUE;
else // AUTDZB
   if n != 31 then UNDEFINED;
```

### Assembler Symbols

*\( <Xd> \)*  
Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

*\( <Xn|SP> \)*  
Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

### Operation

```plaintext
if source_is_sp then
   X[d] = AuthDB(X[d], SP());
else
   X[d] = AuthDB(X[d], X[n]);
```

---

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

Authenticate Instruction address, using key A. This instruction authenticates an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by <Xd> for AUTIA and AUTIZA.
- In X17, for AUTIA1716.
- In X30, for AUTIASP and AUTIAZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIA.
- The value zero, for AUTIZA and AUTIAZ.
- In X16, for AUTIA1716.
- In SP, for AUTIASP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: Integer and System

**Integer**

(ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 1  | 0  | 0  | Rn | Rd |

**AUTIA (Z == 0)**

AUTIA <Xd>, <Xn|SP>

**AUTIZA (Z == 1 & Rn == 1111)**

AUTIZA <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
  UNDEFINED;
if Z == '0' then // AUTIA
  if n == 31 then source_is_sp = TRUE;
else // AUTIZA
  if n != 31 then UNDEFINED;
```

**System**

(ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | x  | 1  | 1  | 0  | x  | 1  | 1  | 1  | 1  | 1  |

CRm op2
AUTIA1716 (CRm == 0001 && op2 == 100)

AUTIA1716

AUTIASP (CRm == 0011 && op2 == 101)

AUTIASP

AUTIAZ (CRm == 0011 && op2 == 100)

AUTIAZ

```plaintext
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 100' // AUTIAZ
    d = 30;
    n = 31;
  when '0011 101' // AUTIASP
    d = 30;
    source_is_sp = TRUE;
  when '0001 100' // AUTIA1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 010' SEE "PACIB";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 01x' SEE "PACIB";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLR1";
  otherwise SEE "HINT";
```

Assembler Symbols

- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```plaintext
if HavePACExt () then
  if source_is_sp then
    X[d] = AuthIA(X[d], SP[]);
  else
    X[d] = AuthIA(X[d], X[n]);
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

Authenticate Instruction address, using key B. This instruction authenticates an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by <Xd> for AUTIB and AUTIZB.
- In X17, for AUTIB1716.
- In X30, for AUTIBSP and AUTIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIB.
- The value zero, for AUTIZB and AUTIBZ.
- In X16, for AUTIB1716.
- In SP, for AUTIBSP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: Integer and System

### Integer (ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z | 1  | 0  | 1  |

### AUTIB (Z == 0)

AUTIB <Xd>, <Xn|SP>

### AUTIZB (Z == 1 & Rn == 1111)

AUTIZB <Xd>

```plaintext
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTIB
    if n == 31 then source_is_sp = TRUE;
else // AUTIZB
    if n != 31 then UNDEFINED;
```

### System (ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | x  | 1  | 1  | 1  | 1  | x  |

CRm        op2
AUTIB1716 (CRm == 0001 & op2 == 110)

AUTIB1716

AUTIBSP (CRm == 0011 & op2 == 111)

AUTIBSP

AUTIBZ (CRm == 0011 & op2 == 110)

AUTIBZ

integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 110' // AUTIBZ
    d = 30;
    n = 31;
  when '0011 111' // AUTIBSP
    d = 30;
    source_is_sp = TRUE;
  when '0001 110' // AUTIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 010' SEE "PACIB";
  when '0001 100' SEE "AUTIA";
  when '0011 00x' SEE "PACIA";
  when '0011 01x' SEE "PACIB";
  when '0011 10x' SEE "AUTIA";
  when '0000 111' SEE "XPACLRI";

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if HavePACExt () then
  if source_is_sp then
    X[d] = AuthIB(X[d], SP());
  else
    X[d] = AuthIB(X[d], X[n]);

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AXFlag

Convert floating-point condition flags from ARM to external format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from a form representing the result of an ARM floating-point scalar compare instruction to an alternative representation required by some software.

System
(ARMv8.5)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | (0)| (0)| (0)| (0)| (0)| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  |
```

System

```
AXFlag
if !HaveFlagFormatExt() then UNDEFINED;
```

Operation

```
bit N = '0';
bz Z = PSTATE.Z OR PSTATE.V;
bz C = PSTATE.C AND NOT(PSTATE.V);
bz V = '0';

PSTATE.N = N;
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = V;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
B.cond

Branch conditionally to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 0  | 1  | 0  | imm19 | 0 | cond |

19-bit signed PC-relative branch offset

B.<cond> <label>

```
bits(64) offset = SignExtend(imm19:'00', 64);
bits(4) condition = cond;
```

Assembler Symbols

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
if ConditionHolds(condition) then
    BranchTo(PC[] + offset, BranchType_DIR);
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

26-bit signed PC-relative branch offset

```
B <label>

BranchType branch_type = if op == '1' then BranchType_DIRCALL else BranchType_DIR;
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

```
<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.
```

Operation

```
if branch_type == BranchType_DIRCALL then X[30] = PC[] + 4;
BranchTo(PC[] + offset, branch_type);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BCAX

Bit Clear and Exclusive OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

Advanced SIMD
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1 |

Advanced SIMD

BCAX <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

if !HaveSHA3Ext () then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va> Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

AArch64.CheckFPAdvSIMDEnabled () ;

bits(128) Vm = V[m];
browse(128) Vn = V[n];
browse(128) Va = V[a];
V[d] = Vn EOR (Vm AND NOT(Va));

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00be8_re3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFC

Bitfield Clear sets a bitfield of <width> bits at bit position <lsb> of the destination register to zero, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

• The encodings in this description are named to match the encodings of BFM.
• The description of BFM gives the operational pseudocode for this instruction.

Leaving other bits unchanged

(ARMv8.2)

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn</td>
</tr>
</tbody>
</table>

32-bit (sf == 0 & N == 0)

BFC <Wd>, #<lsb>, #<width>

is equivalent to

BFM <Wd>, WZR, #(-<lsb> MOD 32), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit (sf == 1 & N == 1)

BFC <Xd>, #<lsb>, #<width>

is equivalent to

BFM <Xd>, XZR, #(-<lsb> MOD 64), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<lsb> For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**BFI**

Bitfield Insert copies a bitfield of \(<width>\) bits from the least significant bits of the source register to bit position \(<lsb>\) of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | sf | 0  | 1  | 0  | 0  | 1  | 0  | N | immr | imms | != 11111 | Rd |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---
Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly. If \(<\text{imms}>\) is greater than or equal to \(<\text{immr}>\), this copies a bitfield of \(<\text{imms}>-<\text{immr}>+1\) bits starting from bit position \(<\text{immr}>\) in the source register to the least significant bits of the destination register. If \(<\text{imms}>\) is less than \(<\text{immr}>\), this copies a bitfield of \(<\text{imms}>+1\) bits from the least significant bits of the source register to bit position \((\text{regsize}-<\text{immr}>)\) of the destination register, where \(\text{regsize}\) is the destination register size of 32 or 64 bits. In both cases the other bits of the destination register remain unchanged.

This instruction is used by the aliases **BFC**, **BFI**, and **BFXIL**.

### 32-bit (sf == 0 &\& N == 0)

**BFM** \(<\text{Wd}>, <\text{Wn}>, \#<\text{immr}>, \#<\text{imms}>\)

### 64-bit (sf == 1 &\& N == 1)

**BFM** \(<\text{Xd}>, <\text{Xn}>, \#<\text{immr}>, \#<\text{imms}>\)

```plaintext
define opc 32-bit (sf == 0 &\& N == 0)
BFM <Wd>, <Wn>, \#<immr>, \#<imms>
define opc 64-bit (sf == 1 &\& N == 1)
BFM <Xd>, <Xn>, \#<immr>, \#<imms>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean inzero;
boolean extend;
integer R;
integer S;
bite(datasize) wmask;
bite(datasize) tmask;
case opc of
  when '00' inzero = TRUE;  extend = TRUE;   // SBFM
  when '01' inzero = FALSE; extend = FALSE;   // BPM
  when '10' inzero = TRUE;  extend = FALSE;   // UBFM
  when '11' UNDEFINED;
if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;
R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

### Assembler Symbols

**<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

**<Wn>** Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

**<Xn>** Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

**<immr>** For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

**<imms>** For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.
### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFC</td>
<td>Rn == '11111' &amp;&amp; UInt(imms) &lt; UInt(immr)</td>
</tr>
<tr>
<td>BFI</td>
<td>Rn != '11111' &amp;&amp; UInt(imms) &lt; UInt(immr)</td>
</tr>
<tr>
<td>BFXIL</td>
<td>UInt(imms) &gt;= UInt(immr)</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(datasize) dst = if inzero then Zeros() else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BFXIL

Bitfield Extract and Insert Low copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

32-bit (sf == 0 && N == 0)

BFXIL <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

BFM <Wd>, <Wn>, #<lsb>, #{<lsb>+<width>-1}

and is the preferred disassembly when UInt(imms) >= UInt(immr).

64-bit (sf == 1 && N == 1)

BFXIL <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

BFM <Xd>, <Xn>, #<lsb>, #{<lsb>+<width>-1}

and is the preferred disassembly when UInt(imms) >= UInt(immr).

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <lsb> For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
- For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- <width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
- For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**BIC (vector, immediate)**

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Assembler Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Vd&gt;</code></td>
<td>Is the name of the SIMD&amp;FP register, encoded in the &quot;Rd&quot; field.</td>
</tr>
</tbody>
</table>
| `<T>`  | For the 16-bit variant: is an arrangement specifier, encoded in “Q”:
|        | 0  | 1  |
|        | 4H | 8H |
| `<imm8>` | Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h". |
| `<amount>` | For the 16-bit variant: is the shift amount encoded in “cmode<1>”:
|        | 0  | 1  |
|        | 2S | 4S |
defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);

V[rd] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Three registers of the same type

|<Vd> |<T>, |<Vn> |<T>, |<Vm> |<T>
---|---|---|---|---|
integer d = UInt(Rd); integer n = UInt(Rn); integer m = UInt(Rm); integer esize = 8; integer datatype = if Q == '1' then 128 else 64; integer elements = datatype DIV esize;

boolean invert = (size<0> == '1');
LogicalOp op = if size<1> == '1' then LogicalOp ORR else LogicalOp AND;

Assembler Symbols

|<Vd>| Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
|<T>| Is an arrangement specifier, encoded in “Q”:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>&lt;T&gt;</td>
</tr>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

|<Vn>| Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
|<Vm>| Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datatype) operand1 = V[n];
bits(datatype) operand2 = V[m];
bits(datatype) result;

if invert then operand2 = NOT(operand2);
case op of
    when LogicalOp AND
        result = operand1 AND operand2;
    when LogicalOp ORR
        result = operand1 OR operand2;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BIC (shifted register)

Bitwise Bit Clear (shifted register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>shift</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

BIC <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

BIC <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datatise = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;
if sf == '0' &6 imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
**Operation**

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags, performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>shift</th>
<th>1</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

opc

32-bit (sf == 0)

BICS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

BICS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
Operation

bits(datasize) operand1 = $X[n]$;
bits(datasize) operand2 = \texttt{ShiftReg}(m, shift\_type, shift\_amount);

if invert then operand2 = NOT(operand2);

case op of
  when \texttt{LogicalOp\_AND} result = operand1 AND operand2;
  when \texttt{LogicalOp\_ORR} result = operand1 OR operand2;
  when \texttt{LogicalOp\_EOR} result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:\texttt{IsZeroBit}(result):'00';

$X[d]$ = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BIF

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

opc2

Three registers of the same type

BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

VBitOp op;

case opc2 of
when '00' op = VBitOp_VEOR;
when '01' op = VBitOp_VBSL;
when '10' op = VBitOp_VBIT;
when '11' op = VBitOp_VBIF;

Assembler Symbols

| <Vd> | Is the name of the SIMD&FP destination register, encoded in the “Rd” field. |
|<T>   | Is an arrangement specifier, encoded in “Q”: |
|<Vn>  | Is the name of the first SIMD&FP source register, encoded in the “Rn” field. |
|<Vm>  | Is the name of the second SIMD&FP source register, encoded in the “Rm” field. |
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
  when VBitOp_VEOR
    operand1 = V[m];
    operand2 = Zeros();
    operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
    operand2 = operand1;
    operand3 = NOT(V[m]);

V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{tabular}{cccccccccccccc}
\hline
0 & Q & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & \hline
\end{tabular}

\textbf{Three registers of the same type}

\begin{verbatim}
BIT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “Q”:

\begin{tabular}{cc}
\hline
Q & <T> \\
0 & 8B \\
1 & 16B \\
\end{tabular}

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
\end{verbatim}
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
  when VBitOp_VEOR
    operand1 = V[m];
    operand2 = Zeros();
    operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
    operand2 = operand1;
    operand3 = NOT(V[m]);

V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Operational information

If $PSTATE.DIT$ is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

26-bit signed PC-relative branch offset

\[
\text{BL } \langle \text{label} \rangle
\]

\[
\text{BranchType } \text{branch_type} = \text{if op == '1' then BranchType_DIRCALL else BranchType_DIR;}
\]

\[
\text{bits(64) offset = SignExtend(imm26:'00', 64);}
\]

Assembler Symbols

\langle \text{label} \rangle Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

\[
\text{if branch_type == BranchType_DIRCALL then } X[30] = \text{PC[] + 4;}
\]

\[
\text{BranchTo(\text{PC[] + offset}, branch_type};
\]

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00688_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

**Integer**

```
BLR <Xn>

integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UNDEFINED;
elsif pac && !HavePACExt() then
    UNDEFINED;

  case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

  if pac then
    if Z == '0' && m != 31 then
      UNDEFINED;
    if branch_type == BranchType_RET then
      if n != 31 then UNDEFINED;
      n = 30;
      source_is_sp = TRUE;

```

**Assembler Symbols**

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
Operation

bits(64) target = X[n];
case branch_type of
  when BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
  if pac then
    bits(64) modifier = if source_is_sp then SP[] else X[m];
    if use_key_a then
      target = AuthIA(target, modifier);
    else
      target = AuthIB(target, modifier);
  if branch_type == BranchType_INDCALL then X[30] = PC[] + 4;
  BranchTo(target, branch_type);
BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by $<X_n>$, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4. The modifier is:

- In the general-purpose register or stack pointer that is specified by $<X_m|SP>$ for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

Key A is used for BLRAA and BLRAAZ, and key B is used for BLRAB and BLRABZ. If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated. The authenticated address is not written back to the general-purpose register.

<table>
<thead>
<tr>
<th>Integer (ARMv8.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>1 1 0 1 0 1 1 Z 0 0 1 1 1 1 1 1 0 0 0 0 1 M Rn Rm</td>
</tr>
<tr>
<td>op A</td>
</tr>
</tbody>
</table>

Key A, zero modifier ($Z == 0 \&\& M == 0 \&\& Rm == 11111$)

```assembly
BLRAAZ <Xn>
```

Key A, register modifier ($Z == 1 \&\& M == 0$)

```assembly
BLRAA <Xn>, <Xm|SP>
```

Key B, zero modifier ($Z == 0 \&\& M == 1 \&\& Rm == 11111$)

```assembly
BLRABZ <Xn>
```

Key B, register modifier ($Z == 1 \&\& M == 1$)

```assembly
BLRAB <Xn>, <Xm|SP>
```

```assembly
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') \&\& (m == 31));
if !pac \&\& m != 0 then
  UNDEFINED;
elseif pac \&\& !HavePACExt() then
  UNDEFINED;

case op of
  when '00' branch_type = BranchType_INDIR;
  when '01' branch_type = BranchType_INDCELL;
  when '10' branch_type = BranchType_RET;
  otherwise UNDEFINED;
if pac then
  if Z == '0' \&\& m != 31 then
    UNDEFINED;
  if branch_type == BranchType_RET then
    if n != 31 then UNDEFINED;
    n = 30;
    source_is_sp = TRUE;
```

BLRAA, BLRAAZ, BLRAB, BLRABZ

Page 105
Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

```plaintext
bits(64) target = X[n];
case branch_type of
  when BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
  if pac then
    bits(64) modifier = if source_is_sp then SP[] else X[m];
    if use_key_a then
      target = AuthIA(target, modifier);
    else
      target = AuthIB(target, modifier);
  if branch_type == BranchType_INDCALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Z  op A  M  Rn  Rm
```

**Integer**

```
BR <Xn>

integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then UNDEFINED;
elsif pac && !HavePACExt() then UNDEFINED;
case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;
if pac then
    if Z == '0' && m != 31 then UNDEFINED;
    if branch_type == BranchType_RET then
        if n != 31 then UNDEFINED;
        n = 30;
        source_is_sp = TRUE;
```

**Assembler Symbols**

`<Xn>` Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
Operation

```plaintext
bits(64) target = X[n];
case branch_type of
  when BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
  if pac then
    bits(64) modifier = if source_is_sp then SP[] else X[m];
    if use_key_a then
      target = AuthIA(target, modifier);
    else
      target = AuthIB(target, modifier);
  if branch_type == BranchType_INDCALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

Key A is used for BRAA and BRAAZ, and key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

### Integer (ARMv8.3)

<table>
<thead>
<tr>
<th>op</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 1</td>
<td>Z 0 0 0 1 1 1 1 0 0 0 0 1 M</td>
</tr>
</tbody>
</table>

#### Key A, zero modifier (Z == 0 && M == 0 && Rm == 11111)

**BRAAZ <Xn>**

#### Key A, register modifier (Z == 1 && M == 0)

**BRAA <Xn>, <Xm|SP>**

#### Key B, zero modifier (Z == 0 && M == 1 && Rm == 11111)

**BRABZ <Xn>**

#### Key B, register modifier (Z == 1 && M == 1)

**BRAB <Xn>, <Xm|SP>**

```plaintext
ingter m = UInt(Rn);
BranchType branch_type;
ingter m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UNDEFINED;
elsif pac && !HavePACExt() then
    UNDEFINED;
end case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;
    if branch_type == BranchType_RET then
        if n != 31 then UNDEFINED;
        n = 30;
        source_is_sp = TRUE;
```

BRAA, BRAAZ, BRAB, BRABZ
<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

```
bits(64) target = X[n];
case branch_type of
    when BranchType_INDIR
        if InGuardedPage then
            if n == 16 || n == 17 then
                BTypeNext = '01';
            else
                BTypeNext = '11';
            else
                BTypeNext = '01';
        when BranchType_INDCALL
            BTypeNext = '10';
        when BranchType_RET
            BTypeNext = '00';
    if pac then
        bits(64) modifier = if source_is_sp then SP[] else X[m];
        if use_key_a then
            target = AuthIA(target, modifier);
        else
            target = AuthIB(target, modifier);
        if branch_type == BranchType_INDCALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Breakpoint instruction. A BRK instruction generates a Breakpoint Instruction exception. The PE records the exception in $ESR_{ELx}$, using the EC value 0x3c, and captures the value of the immediate argument in $ESR_{ELx.ISS}$.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**System**

```language=asm
BRK #<imm>
```

bit(16) comment = imm16;
if HaveBTEExt() then
  BTypeCompatible = TRUE;

**Assembler Symbols**

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

**Operation**

```language=asm
AArch64.SoftwareBreakpoint(comment);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00b08_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

BSL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;

Assembler Symbols

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “Q”:

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
  when VBitOp_VEOR
    operand1 = V[m];
    operand2 = Zeros();
    operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
    operand2 = operand1;
    operand3 = NOT(V[m]);

V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BTI

Branch Target Identification. A BTI instruction is used to guard against the execution of instructions which are not the intended target of a branch.

Outside of a guarded memory region, a BTI instruction executes as a NOP. Within a guarded memory region while \( PSTATE.BTYPE \neq 0b00 \), a BTI instruction compatible with the current value of PSTATE.BTYPE will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region.

The operand \(<\text{targets}>\) passed to a BTI instruction determines the values of \( PSTATE.BTYPE \) which the BTI instruction is compatible with.

Within a guarded memory region, while \( PSTATE.BTYPE \neq 0b00 \), all instructions will generate a Branch Target Exception, other than BRK, BTI, HLT, PACIASP, and PACIBSP, which may not. See the individual instructions for details.

System
(ARMv8.5)

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CRm  op2

System

BTI {<targets>}

```

```
SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111'
      SEE "XPACLRI";
  when '0001 xxx'
      SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
      if !HaveRASExt() then EndOfInstruction();           // Instruction executes as NOP
      op = SystemHintOp_ESB;
  when '0010 001'
      if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
      op = SystemHintOp_PSB;
  when '0010 100'
      if !HaveSelfHostedTrace() then EndOfInstruction();   // Instruction executes as NOP
      op = SystemHintOp_TSB;
  when '0011 xxx'
      SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xxO'
      op = SystemHintOp_BT;
      BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction();                           // Instruction executes as NOP
```

Assembler Symbols

\(<\text{targets}>\) Is the type of indirection, encoded in "op2<2:1>":
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, TRUE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, TRUE);
        WaitForEvent();
  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, FALSE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, FALSE);
        WaitForInterrupt();
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    BTypeNext = '00';
  otherwise // do nothing

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASA and CASAL load from memory with acquire semantics.
- CASL and CASAL store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails. If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, or <Xs>, is restored to the value held in the register before the instruction was executed.

**No offset**

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | L  | 1  | Rs | o0 | 1  | 1  | 1  | 1  | Rn | |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size
32-bit CAS (size == 10 & L == 0 & o0 == 0)

CAS <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASA (size == 10 & L == 1 & o0 == 0)

CASA <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASAL (size == 10 & L == 1 & o0 == 1)

CASAL <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASL (size == 10 & L == 0 & o0 == 1)

CASL <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit CAS (size == 11 & L == 0 & o0 == 0)

CAS <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASA (size == 11 & L == 1 & o0 == 0)

CASA <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASAL (size == 11 & L == 1 & o0 == 1)

CASAL <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASL (size == 11 & L == 0 & o0 == 1)

CASL <Xs>, <Xt>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldaccctype = if L == '1' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;
AccType staccctype = if o0 == '1' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, datasize DIV 8, stacctype] = newvalue;
X[s] = ZeroExtend(data, regsize)

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails. If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset
(ARMv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 1 0 0 0 1 1 1 1 1 | Rs | 0 0 1 1 1 1 | Rn | Rt |

size

CASB (L == 1 && o0 == 0)

CASB <Ws>, <Wt>, [<Xn|SP>{}, #0]]

CASALB (L == 1 && o0 == 1)

CASALB <Ws>, <Wt>, [<Xn|SP>{}, #0]]

CASB (L == 0 && o0 == 0)

CASB <Ws>, <Wt>, [<Xn|SP>{}, #0]]

CASLB (L == 0 && o0 == 1)

CASLB <Ws>, <Wt>, [<Xn|SP>{}, #0]]

if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

Assembler Symbols

<Ws> is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, datasize DIV 8, stacctype] = newvalue;
X[s] = ZeroExtend(data, regsize);
CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset

(ARMv8.1)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | L  | 1  | Rs | o0 | 1  | 1  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

size

CASAH (L == 1 && o0 == 0)

CASAH <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASALH (L == 1 && o0 == 1)

CASALH <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASH (L == 0 && o0 == 0)

CASH <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASLH (L == 0 && o0 == 1)

CASLH <Ws>, <Wt>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, datasize DIV 8, stacctype] = newvalue;

X[s] = ZeroExtend(data, regsize);
CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- CASPA and CASPAL load from memory with acquire semantics.
- CASPL and CASPAL store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails. If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is <Ws> and <W(s+1)>, or <Xs> and <X(s+1)>, are restored to the values held in the registers before the instruction was executed.

**No offset**

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | sz| 0  | 0  | 1  | 0  | 0  | 0  | 0  | L  | 1  | Rs | o0 | 1  | 1  | 1  | 1  | 1  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Rt2
32-bit CASP (sz == 0 & L == 0 & o0 == 0)

CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPA (sz == 0 & L == 1 & o0 == 0)

CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPAL (sz == 0 & L == 1 & o0 == 1)

CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPL (sz == 0 & L == 0 & o0 == 1)

CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

64-bit CASP (sz == 1 & L == 0 & o0 == 0)

CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPA (sz == 1 & L == 1 & o0 == 0)

CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPAL (sz == 1 & L == 1 & o0 == 1)

CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPL (sz == 1 & L == 0 & o0 == 1)

CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 32 << UInt(sz);
integer regsize = datasize;
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

Assembler Symbols

<Ws>  Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.

<W(s+1)> Is the 32-bit name of the second general-purpose register to be compared and loaded.

<Wt>  Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Wt> must be an even-numbered register.

<W(t+1)> Is the 32-bit name of the second general-purpose register to be conditionally stored.

<Xs>  Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.

<X(s+1)> Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt>  Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
<X(t+1)> Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s];
bits(datasize) s2 = X[s+1];
bits(datasize) t1 = X[t];
bits(datasize) t2 = X[t+1];

comparevalue = if BigEndian() then s1:s2 else s2:s1;
newvalue = if BigEndian() then t1:t2 else t2:t1;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);
if n == 31 then
    CheckSPAlignment();
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, (2 * datasize) DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, (2 * datasize) DIV 8, stacctype] = newvalue;

if BigEndian() then
    X[s] = ZeroExtend(data<2*datasize-1:datasize>, regsize);
    X[s+1] = ZeroExtend(data<datasize-1:0>, regsize);
else
    X[s] = ZeroExtend(data<datasize-1:0>, regsize);
    X[s+1] = ZeroExtend(data<2*datasize-1:datasize>, regsize);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>imm19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

```
CBNZ <Wt>, <label>
```

**64-bit (sf == 1)**

```
CBNZ <Xt>, <label>
```

integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);

### Assembler Symbols

- `<Wt>`: The 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- `<Xt>`: The 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- `<label>`: The program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

### Operation

```
bits(datasize) operand1 = X[t];
if IsZero(operand1) == iszero then
    BranchTo(PC[] + offset, BranchType_DIR);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 0  | 1  | 1  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| op |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| imm19 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rt  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (sf == 0)

```c
CBZ <Wt>, <label>
```

64-bit (sf == 1)

```c
CBZ <Xt>, <label>
```

```c
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- `<Xt>`: Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- `<label>`: Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```c
bits(datasize) operand1 = X[t];
if IsZero(operand1) == iszero then
   BranchTo(PC[] + offset, BranchType_DIR);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_re3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Conditional Compare Negative (immediate) sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

### Assembler Symbols

- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.
- `<imm>`: Is a five bit unsigned (positive) immediate encoded in the “imm5” field.
- `<nzcv>`: Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.
- `<cond>`: Is one of the standard conditions, encoded in the “cond” field in the standard way.

### Operation

1. integer `n = UInt(Rn);`
2. integer `datasize = if sf == '1' then 64 else 32;`
3. boolean `sub_op = (op == '1');`
4. bits(4) `condition = cond;`
5. bits(4) `flags = nzcv;`
6. bits(datasize) `imm = ZeroExtend(imm5, datasize);`

```plaintext
if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (~, flags) = AddWithCarry(operand1, operand2, carry_in);
    PSTATE.<N,Z,C,V> = flags;
```

### Operational Information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CCMN (register)

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| op |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (sf == 0)

CCMN `<Wn>`, `<Wm>`, #<nzcv>, <cond>

64-bit (sf == 1)

CCMN `<Xn>`, `<Xm>`, #<nzcv>, <cond>

integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;

Assembler Symbols

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<nzcv>` Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
`<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';

if ConditionHolds(condition) then
  if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
  (-, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CCMP (immediate)

Conditional Compare (immediate) sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>imm5</th>
<th></th>
<th></th>
<th>cond</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>0</th>
<th>nzcv</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CCMP <Wn>, #<imm>, #<nzcv>, <cond>

64-bit (sf == 1)

CCMP <Xn>, #<imm>, #<nzcv>, <cond>

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler Symbols

- `<Wn>` is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xn>` is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<imm>` is a five bit unsigned (positive) immediate encoded in the "imm5" field.
- `<nzcv>` is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
- `<cond>` is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (~, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CCMP (register)

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

```
CCMP <Wn>, <Wm>, #<nzcv>, <cond>
```

64-bit (sf == 1)

```
CCMP <Xn>, <Xm>, #<nzcv>, <cond>
```

Assembler Symbols

- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<nzcv>` Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
- `<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```java
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, carry_in);
    PSTATE.<N,Z,C,V> = flags;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CFINV

Invert Carry Flag. This instruction inverts the value of the PSTATE.C flag.

System (ARMv8.4)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  |
```

**CRm**

System

```java
if !HaveFlagManipulateExt() then UNDEFINED;
```

Operation

```java
PSTATE.C = NOT(PSTATE.C);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Control Flow Prediction Restriction by Context prevents control flow predictions that predict execution addresses, based on information gathered from earlier execution within a particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see CFP RCTX, Control Flow Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  |

Rt
L  op1  CRn  CRm  op2
```

System

CFP RCTX, <Xt>

is equivalent to

SYS #3, c8, c3, #4, <Xt>

and is always the preferred disassembly.

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.
CINC

Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSINC. This means:

• The encodings in this description are named to match the encodings of CSINC.
• The description of CSINC gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CINC <Wd>, <Wn>, <cond>

is equivalent to

CSINC <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

CINC <Xd>, <Xn>, <cond>

is equivalent to

CSINC <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

<cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

CINV <Wd>, <Wn>, <cond>

is equivalent to

CSINV <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

### 64-bit (sf == 1)

CINV <Xd>, <Xn>, <cond>

is equivalent to

CSINV <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

**Assembler Symbols**

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- `<cond>` is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

**Operation**

The description of CSINV gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CLREX

Clear Exclusive clears the local monitor of the executing PE.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | CRm | 0  | 1  | 0  | 1  | 1  | 1  | 1  |

System

CLREX {#<imm>}

// CRm field is ignored

Assembler Symbols

<imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

**ClearExclusiveLocal** (*ProcessorID()*);
CLS (vector)

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The count does not include the most significant bit itself.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 1 1 1 0 0 0 0 0 0 1 0 1 0 0 0
size 1 Rn Rd
```

Vector

$CLS\ <Vd>\ .<T>, <Vn> .<T>$

integer $d = \text{UInt}(Rd);$  
integer $n = \text{UInt}(Rn);$  

if $size == '11'$ then UNDEFINED;  
integer $esize = 8 << \text{UInt}(size);$  
integer $datasize = if $Q == '1'$ then 128 else 64;  
integer $elements = \text{datasize DIV esize};$

```
CountOp countop = if $U == '1'$ then CountOp CLZ else CountOp CLS;
```

Assembler Symbols

$<Vd>$ 
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ 
Is an arrangement specifier, encoded in “size:Q”:

```
size Q <T>   
00 0 8B     
00 1 16B    
01 0 4H     
01 1 8H     
10 0 2S     
10 1 4S     
11 x RESERVED
```

$<Vn>$ 
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = $V[n];$
bits(datasize) result;

integer count;
for $e = 0$ to elements-1  
  if countop == CountOp CLS then  
    count = CountLeadingSignBits(Elem[operand, e, esize]);  
  else  
    count = CountLeadingZeroBits(Elem[operand, e, esize]);  
  Elem[result, e, esize] = count<esize-1:0>;
$V[d] = result;
```

Operational information

If $PSTATE.DIT$ is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ▫ The values of the data supplied in any of its registers.
  ▫ The values of the NZCV flags.
Count Leading Sign bits counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | Rd |
| op |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (sf == 0)

CLS <Wd>, <Wn>

64-bit (sf == 1)

CLS <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
CountOp opcode = if op == '0' then CountOp_CLZ else CountOp_CLS;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

integer result;
bits(datasize) operand1 = X[n];

if opcode == CountOp_CLZ then
  result = CountLeadingZeroBits(operand1);
else
  result = CountLeadingSignBits(operand1);

X[d] = result<datasize-1:0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CLZ (vector)

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

```assembly
CLZ <Vd>.<T>, <Vn>.<T>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```assembly
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
  if countop == CountOp_CLS then
    count = CountLeadingSignBits(Elem[operand, e, esize]);
  else
    count = CountLeadingZeroBits(Elem[operand, e, esize]);
  Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
CLZ

Count Leading Zeros counts the number of binary zero bits before the first binary one bit in the value of the source register, and writes the result to the destination register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |

32-bit (sf == 0)

CLZ <Wd>, <Wn>

64-bit (sf == 1)

CLZ <Xd>, <Xn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
CountOp opcode = if op == '0' then CountOp_CLZ else CountOp_CLS;
```

Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```plaintext
integer result;
bits(datasize) operand1 = X[n];
if opcode == CountOp_CLZ then
    result = CountLeadingZeroBits(operand1);
else
    result = CountLeadingSignBits(operand1);
X[d] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>Rm</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rn</th>
<th></th>
<th></th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar

CMEQ <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');

Vector

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>Rm</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rn</th>
<th></th>
<th></th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector

CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');

Assembler Symbols

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;V&gt;</td>
<td>Is a width specifier, encoded in &quot;size&quot;:</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>&lt;V&gt;</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size-Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMEQ (zero)

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 1   | 1   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 1   | 1   | 0   | Rn  | Rd  |

U

op

Scalar

CMEQ <V><d>, <V><n>, #0

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
  case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | Q   | 1   | 1   | 1   | 0   | size | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 1   | 1   | 0   | Rn  | Rd  |

U

op

Vector

CMEQ <Vd>.<T>, <Vn>.<T>, #0

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
  case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<V> Is a width specifier, encoded in "size":

CMEQ (zero)
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bots(datasize) operand = V[n];
bots(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
    when CompareOp_GT test_passed = element > 0;
    when CompareOp_GE test_passed = element >= 0;
    when CompareOp_EQ test_passed = element == 0;
    when CompareOp_LE test_passed = element <= 0;
    when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Scalar

CMGE <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Vector

CMGE <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |

U | op

**Scalar**

CMGE \(<V><d>, <V><n>, #0\)

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |

U | op

**Vector**

CMGE \(<Vd>.<T>, <Vn>.<T>, #0\)

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Assembler Symbols

\(<V>\) Is a width specifier, encoded in "size":

CMGE (zero)
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n>
Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
  element = SInt(Elem[operand, e, esize]);
  case comparison of
    when CompareOp_GT test_passed = element > 0;
    when CompareOp_GE test_passed = element >= 0;
    when CompareOp_EQ test_passed = element == 0;
    when CompareOp_LE test_passed = element <= 0;
    when CompareOp_LT test_passed = element < 0;
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGT (register)

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
size 1 Rm 0 0 1 1 0 1 Rn Rd

U eq

Scalar

CMGT <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
size 1 Rm 0 0 1 1 0 1 Rn Rd

U eq

Vector

CMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer size:Q = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```cpp
CheckFPAdvSIMDEnabled64();
b_bits(datasize) operand1 = V[n];
b_bits(datasize) operand2 = V[m];
b_bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 < element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGT (zero)

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  |

U  | op | Rd | Rn |
```

Vector

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  |

U  | op | Rd | Rn |
```

Assembler Symbols

\(<V>\) Is a width specifier, encoded in "size":

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;
<d>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n>
Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMHI (register)

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

Scalar

CMHI <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

Vector

CMHI <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMHS (register)

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |    |    | size| 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| eq|

Scalar

CMHS <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 6 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 0  | Q | 1  | 0  | 1  | 1  | 1  | 0  |    |    | size| 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| eq|

Vector

CMHS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m =UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Assembler Symbols

| <V> | Is a width specifier, encoded in “size”:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;d&gt;</th>
<th>Is the number of the SIMD&amp;FP destination register, in the &quot;Rd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;n&gt;</td>
<td>Is the number of the first SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
</tbody>
</table>
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size-Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bites(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMLE (zero)

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
|   | 1 | 1 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
```

**Scalar**

\[ \text{CMLE } <V><d>, <V><n>, #0} \]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

**Vector**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
|   | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
```

**Vector**

\[ \text{CMLE } <Vd>.<T>, <Vn>.<T>, #0} \]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

**Assembler Symbols**

\[ <V> \]

Is a width specifier, encoded in "size": 
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n>
Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
  element = SInt(Elem[operand, e, esize]);
  case comparison of
    when CompareOp_GT test_passed = element > 0;
    when CompareOp_GE test_passed = element >= 0;
    when CompareOp_EQ test_passed = element == 0;
    when CompareOp_LE test_passed = element <= 0;
    when CompareOp_LT test_passed = element < 0;
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMLT (zero)

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: `Scalar` and `Vector`.

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | Rd |

Scalar

```
CMLT <V><d>, <V><n>, #0
```

```plaintext
text
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | Rd |

Vector

```
CMLT <Vd>.<T>, <Vn>.<T>, #0
```

```plaintext
text
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Assembler Symbols

```
<V>
Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

```
<d>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
```

```
<n>
Is the number of the SIMD&FP source register, encoded in the "Rn" field.
```

```
<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
```

```
<T>
Is an arrangement specifier, encoded in “size:Q”:
```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bites(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
case comparison of
    when CompareOp_GT test_passed = element > 0;
    when CompareOp_GE test_passed = element >= 0;
    when CompareOp_EQ test_passed = element == 0;
    when CompareOp_LE test_passed = element <= 0;
    when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMN (extended register)

Compare Negative (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the \(<Rm>\) register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (extended register). This means:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMN <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

is equivalent to

ADDS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

and is always the preferred disassembly.

64-bit (sf == 1)

CMN <Xn|SP>, <R><m>{, <extend> {#<amount>}}

is equivalent to

ADDS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

and is always the preferred disassembly.

Assembler Symbols

- \(<Wn|WSP>\) is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- \(<Wm>\) is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xn|SP>\) is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- \(<R>\) is a width specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>(&lt;R&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

- \(<m>\) is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- \(<extend>\) For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>(&lt;extend&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '111111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

32-bit register
For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

**Operation**

The description of [ADDS (extended register)](#) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMN (immediate)

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This is an alias of \texttt{ADDS (immediate)}. This means:

- The encodings in this description are named to match the encodings of \texttt{ADDS (immediate)}.
- The description of \texttt{ADDS (immediate)} gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>imm12</th>
<th>Rn</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

\texttt{CMN <Wn|WSP>, \#<imm>{, <shift>}}

is equivalent to

\texttt{ADDS WZR, <Wn|WSP>, \#<imm>{, <shift>}}

and is always the preferred disassembly.

64-bit (sf == 1)

\texttt{CMN <Xn|SP>, \#<imm>{, <shift>}}

is equivalent to

\texttt{ADDS XZR, <Xn|SP>, \#<imm>{, <shift>}}

and is always the preferred disassembly.

**Assembler Symbols**

- \texttt{<Wn|WSP>} is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- \texttt{<Xn|SP>} is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- \texttt{<imm>} is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- \texttt{<shift>} is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

<table>
<thead>
<tr>
<th>shift&lt;0&gt;</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

**Operation**

The description of \texttt{ADDS (immediate)} gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**CMN (shifted register)**

Compare Negative (shifted register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of *ADDS (shifted register)*. This means:

- The encodings in this description are named to match the encodings of *ADDS (shifted register)*.
- The description of *ADDS (shifted register)* gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

```
CMN <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS WZR, <Wn>, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

**64-bit (sf == 1)**

```
CMN <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS XZR, <Xn>, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:
  
<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

The description of *ADDS (shifted register)* gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
CMP (extended register)

Compare (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the \(<Rm>\) register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (extended register). This means:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

\[\text{CMP} \,<\text{Wm}\>, \,<\text{Wm}\>{\text{,}, \,<\text{extend}\>{\text{,}#\langle\text{amount}\rangle}}}\]

is equivalent to

\[\text{SUBS} \,\text{WZR}, \,<\text{Wm}\>{\text{,}, \,<\text{extend}\>{\text{,}#\langle\text{amount}\rangle}}}\]

and is always the preferred disassembly.

### 64-bit (sf == 1)

\[\text{CMP} \,<\text{SP}\>, \,<\text{Rm}\>{\text{,}, \,<\text{extend}\>{\text{,}#\langle\text{amount}\rangle}}}\]

is equivalent to

\[\text{SUBS} \,\text{XZR}, \,<\text{SP}\>, \,<\text{Rm}\>{\text{,}, \,<\text{extend}\>{\text{,}#\langle\text{amount}\rangle}}}\]

and is always the preferred disassembly.

### Assembler Symbols

- \(<\text{Wm}\>) Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- \(<\text{Wm}\>) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<\text{SP}\>) Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- \(<\text{R}>\) Is a width specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>(&lt;\text{R}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

- \(<\text{m}>\) Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- \(<\text{extend}>\) For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>(&lt;\text{extend}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LS</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases \(<\text{extend}>\) is required and must be UXTW when "option" is '010'.

CMP (extended register)
For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMP (immediate)

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (immediate). This means:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMP <Wn|WSP>, #<imm>{, <shift>}

is equivalent to

SUBS WZR, <Wn|WSP>, #<imm> {, <shift>}

and is always the preferred disassembly.

64-bit (sf == 1)

CMP <Xn|SP>, #<imm>{, <shift>}

is equivalent to

SUBS XZR, <Xn|SP>, #<imm> {, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

<table>
<thead>
<tr>
<th>shift&lt;0&gt;</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMP (shifted register)

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMP \(<Wn>, <Wm>\{, <shift> \#<amount>\}

is equivalent to

SUBS WZR, <Wn>, <Wm> \{, <shift> \#<amount>\}

and is always the preferred disassembly.

64-bit (sf == 1)

CMP \(<Xn>, <Xm>\{, <shift> \#<amount>\}

is equivalent to

SUBS XZR, <Xn>, <Xm> \{, <shift> \#<amount>\}

and is always the preferred disassembly.

Assembler Symbols

\(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
\(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
\(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
\(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
\(<shift>\) Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<amount>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ▫ The values of the data supplied in any of its registers.
  ▫ The values of the NZCV flags.
CMPP

Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This is an alias of SUBPS. This means:

- The encodings in this description are named to match the encodings of SUBPS.
- The description of SUBPS gives the operational pseudocode for this instruction.

### Integer (ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

- **Xm**
- **Xn**
- **Xd**

### Integer

CMPP `<Xn|SP>, <Xm|SP>` is equivalent to

SUBPS XZR, `<Xn|SP>, <Xm|SP>`

and is always the preferred disassembly.

### Assembler Symbols

- `<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<Xm|SP>` Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

### Operation

The description of SUBPS gives the operational pseudocode for this instruction.
CMTST

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 0 | 1 | 1 | Rn | Rd |

Scalar

```
CMTST <V><d>, <V><n>, <V><m>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 0 | 1 | 1 | Rn | Rd |

Vector

```
CMTST <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');

Assembler Symbols

```
<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size-Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;
for e = 0 to elements-1
   element1 = Elem[operand1, e, esize];
   element2 = Elem[operand2, e, esize];
   if and_test then
      test_passed = !IsZero(element1 AND element2);
   else
      test_passed = (element1 == element2);
   Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CNEG

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSNEG. This means:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CNEG <Wd>, <Wn>, <cond>

is equivalent to

CSNEG <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

CNEG <Xd>, <Xn>, <cond>

is equivalent to

CSNEG <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

- <Wd> is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <cond> is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSNEG gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\[
\begin{array}{cccccccccccc}
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & Rn & Rd
\end{array}
\]

**Vector**

\[
\text{CNT } <Vd>..<T>, <Vn>..<T>
\]

\[
\begin{align*}
\text{integer } d &= \text{ UInt}(Rd); \\
\text{integer } n &= \text{ UInt}(Rn); \\
\text{if } size \neq '00' \text{ then UNDEFINED;} \\
\text{integer } esize &= 8; \\
\text{integer } datasize &= \text{ if } Q = '1' \text{ then 128 else 64;} \\
\text{integer } elements &= \text{ datasize DIV 8;}
\end{align*}
\]

**Assembler Symbols**

\[
\begin{array}{ccc}
\text{size} & \text{Q} & \text{T} \\
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & x & \text{RESERVED} \\
1x & x & \text{RESERVED}
\end{array}
\]

\[
\text{<Vd>} \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.} \\
\text{<T>} \quad \text{Is an arrangement specifier, encoded in "size:Q":}
\]

\[
\begin{align*}
\text{CheckFPAdvSIMDEnabled64}(); \\
\text{bits(datasize) operand} &= \text{ V}[n]; \\
\text{bits(datasize) result}; \\
\text{integer count}; \\
\text{for } e = 0 \text{ to elements-1} \\
\quad \text{count} &= \text{ BitCount(Elem[operand, e, esize]);} \\
\quad \text{Elem[result, e, esize]} &= \text{ count<esize-1:0>;} \\
\text{V[d]} &= \text{ result;}
\end{align*}
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions, based on information gathered from earlier execution within a particular execution context, from allowing later speculative execution within that context to be observable through side-channels. For more information, see **CPP RCTX, Cache Prefetch Prediction Restriction by Context**.

This is an alias of **SYS**. This means:
- The encodings in this description are named to match the encodings of **SYS**.
- The description of **SYS** gives the operational pseudocode for this instruction.

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  1 1 0 1 0 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 1 1
  \( L \quad \text{op1} \quad \text{CRn} \quad \text{CRm} \quad \text{op2} \)
```

**System**

**CPP RCTX, \(<\text{Xt}\>)**

is equivalent to

**SYS #3, C8, C3, #7, \(<\text{Xt}\>)**

and is always the preferred disassembly.

**Assembler Symbols**

\(<\text{Xt}\>) is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

**Operation**

The description of **SYS** gives the operational pseudocode for this instruction.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CRC32B, CRC32H, CRC32W, CRC32X

The CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In ARMv8-A, this is an optional instruction, and in ARMv8.1 it is mandatory for all implementations to implement it. The ID_AA64ISAR0_ELI.CRC32 indicates whether this instruction is supported.

### Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
- `<Wm>`: Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

### Operation

```plaintext
bits(32) acc = X[n]; // accumulator
bits(size) val = X[m]; // input value
bits(32) poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>

bits(32+size) tempacc = BitReverse(acc) : Zeros(size);
bits(size+32) tempval = BitReverse(val) : Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

### CRC32B (sf == 0 & sz == 00)

- `CRC32B <Wd>, <Wn>, <Wm>`

### CRC32H (sf == 0 & sz == 01)

- `CRC32H <Wd>, <Wn>, <Wm>`

### CRC32W (sf == 0 & sz == 10)

- `CRC32W <Wd>, <Wn>, <Wm>`

### CRC32X (sf == 1 & sz == 11)

- `CRC32X <Wd>, <Wn>, <Xm>`

If !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);  // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In ARMv8-A, this is an **OPTIONAL** instruction, and in ARMv8.1 it is mandatory for all implementations to implement it. **ID_AA64ISAR0_EL1**.CRC32 indicates whether this instruction is supported.

| sf | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | sz | 0 | 1 | 0 | 1 | 1 | Rm | 0 | 1 | 0 | 1 | Rn | Rd | C |

CRC32CB (sf == 0 && sz == 00)

CRC32CB <Wd>, <Wn>, <Wm>

CRC32CH (sf == 0 & & sz == 01)

CRC32CH <Wd>, <Wn>, <Wm>

CRC32CW (sf == 0 & & sz == 10)

CRC32CW <Wd>, <Wn>, <Wm>

CRC32CX (sf == 1 & & sz == 11)

CRC32CX <Wd>, <Wn>, <Xm>

if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);   // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.

<Wm> Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

bits(32) acc = X[n]; // accumulator
bits(size) val = X[m]; // input value
bits(32) poly = {if crc32c then 0x1EDC6F41 else 0x04C11DB7}<31:0>;

bits(32+size) tempacc = BitReverse(acc) : Zeros(size);
bits(size+32) tempval = BitReverse(val) : Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction. No instruction other than branch instructions appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.\{N,Z,C,V\} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE prediction state for any SVE instructions.

For purposes of the definition of CSDB, PSTATE.\{N,Z,C,V\} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.\{N,Z,C,V\} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0 0 1 0 1 0 0 1 1 1 1
```

```
System

CSDB

SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 111'
        SEE "XPACLRI";
    when '0001 xxx'
        SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
    when '0010 000'
        if !HaveRASExt() then EndOfInstruction();
        op = SystemHintOp_ESB;
    when '0010 001'
        if !HaveStatisticalProfiling() then EndOfInstruction();
        op = SystemHintOp_PSE;
    when '0010 010'
        if !HaveSelfHostedTrace() then EndOfInstruction();
        op = SystemHintOp_TSE;
    when '0010 100'
        op = SystemHintOp_CSD;
    when '0011 xxx'
        SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    when '0100 xx0'
        op = SystemHintOp_BTI;
        BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
    otherwise EndOfInstruction();
```

```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, TRUE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, TRUE);
        WaitForEvent();
    when SystemHintOp_WFI
      if !InterruptPending() then
        if PSTATE.EL == EL0 then
          // Check for traps described by the OS which may be EL1 or EL2.
          AArch64.CheckForWFXTrap(EL1, FALSE);
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
          // Check for traps described by the Hypervisor.
          AArch64.CheckForWFXTrap(EL2, FALSE);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
          // Check for traps described by the Secure Monitor.
          AArch64.CheckForWFXTrap(EL3, FALSE);
          WaitForInterrupt();
    when SystemHintOp_SEV
      SendEvent();
    when SystemHintOp_SEVL
      SendEventLocal();
    when SystemHintOp_ESB
      SynchronizeErrors();
      AArch64.ESBOperation();
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
        AArch64.vESBOperation();
        TakeUnmaskedSErrorInterrupts();
    when SystemHintOp_PSB
      ProfilingSynchronizationBarrier();
    when SystemHintOp_TSB
      TraceSynchronizationBarrier();
    when SystemHintOp_CSDB
      ConsumptionOfSpeculativeDataBarrier();
    when SystemHintOp_BTI
      BTypeNext = '00';
  otherwise // do nothing
Conditional Select returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register.

<table>
<thead>
<tr>
<th>sf</th>
<th>Rm</th>
<th>cond</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CSEL <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSEL <Xd>, <Xn>, <Xm>, <cond>

Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<cond>`: Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bite(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');

bits(datasize) result;
bite(datasize) operand1 = X[n];
bite(datasize) operand2 = X[m];
if ConditionHolds(condition) then
  result = operand1;
else
  result = operand2;
  if else_inv then result = NOT(result);
  if else_inc then result = result + 1;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CSET

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| sf | 0 0 1 1 0 0 1 0 0 | 1 1 1 1 1 | != 11x | 0 1 1 1 1 1 | Rd |
```

**32-bit (sf == 0)**

CSET <Wd>, <cond>

is equivalent to

CSINC <Wd>, WZR, WZR, invert(<cond>)

and is always the preferred disassembly.

**64-bit (sf == 1)**

CSET <Xd>, <cond>

is equivalent to

CSINC <Xd>, XZR, XZR, invert(<cond>)

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.
- `<cond>` Is one of the standard conditions, excluding AL and NV, encoded in the “cond” field with its least significant bit inverted.

**Operation**

The description of CSINC gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This is an alias of `CSINV`. This means:

- The encodings in this description are named to match the encodings of `CSINV`.
- The description of `CSINV` gives the operational pseudocode for this instruction.

| sf | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | != 111x | 0 | 0 | 1 | 1 | 1 | 1 | 1 | Rd |
| op | Rm | cond | o2 | Rn |

### 32-bit (sf == 0)

\[
\text{CSETM } \langle Wd \rangle, \langle \text{cond} \rangle
\]

is equivalent to

\[
\text{CSINV } \langle Wd \rangle, \text{WZR, WZR, invert}(\langle \text{cond} \rangle)
\]

and is always the preferred disassembly.

### 64-bit (sf == 1)

\[
\text{CSETM } \langle Xd \rangle, \langle \text{cond} \rangle
\]

is equivalent to

\[
\text{CSINV } \langle Xd \rangle, \text{XZR, XZR, invert}(\langle \text{cond} \rangle)
\]

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<cond>` Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

**Operation**

The description of `CSINV` gives the operational pseudocode for this instruction.

**Operational information**

If `PSTATE.DIT` is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CSINC

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases CINC and CSET.

32-bit (sf == 0)

CSINC <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSINC <Xd>, <Xn>, <Xm>, <cond>

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINC</td>
<td>Rm != '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn != '11111' &amp;&amp; Rn == Rm</td>
</tr>
<tr>
<td>CSET</td>
<td>Rm == '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');

result = operand1;
if else_inv then result = NOT(result);
if else_inc then result = result + 1;

X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CSINV

Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV, and CSETM.

### 32-bit (sf == 0)

CSINV <Wd>, <Wn>, <Wm>, <cond>

### 64-bit (sf == 1)

CSINV <Xd>, <Xn>, <Xm>, <cond>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');
```

### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINV</td>
<td>Rm != '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn != '11111' &amp;&amp; Rn == Rm</td>
</tr>
<tr>
<td>CSETM</td>
<td>Rm == '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
if ConditionHolds(condition) then
    result = operand1;
else
    result = operand2;
    if else_inv then result = NOT(result);
    else if else_inc then result = result + 1;
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CSNEG

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias CNEG.

32-bit (sf == 0)

CSNEG <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSNEG <Xd>, <Xn>, <Xm>, <cond>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '1');

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNEG</td>
<td>cond != '111' &amp;&amp; Rn == Rm</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
if ConditionHolds(condition) then
  result = operand1;
else
  result = operand2;
  if else_inv then result = NOT(result);
  if else_inc then result = result + 1;
X[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Data Cache operation. For more information, see \texttt{SYS}. This means:

- The encodings in this description are named to match the encodings of \texttt{SYS}.
- The description of \texttt{SYS} gives the operational pseudocode for this instruction.

System

\texttt{DC <dc\_op>, <Xt>}

is equivalent to

\texttt{SYS #<op1>, C7, <Cm>, #<op2>, <Xt>}

and is the preferred disassembly when \texttt{SysOp(op1,'0111',CRm,op2) == Sys\_DC}.

Assembler Symbols

\texttt{<dc\_op>} is a DC instruction name, as listed for the DC system instruction group, encoded in “op1:CRm:op2”:

<table>
<thead>
<tr>
<th>\texttt{op1}</th>
<th>\texttt{CRm}</th>
<th>\texttt{op2}</th>
<th>\texttt{Architectural Feature}</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0110</td>
<td>001</td>
<td>IVAC</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>010</td>
<td>ISW</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>011</td>
<td>IGVAC ARMv8.5-MemTag</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>100</td>
<td>IGSW ARMv8.5-MemTag</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>101</td>
<td>IGDVAC ARMv8.5-MemTag</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>110</td>
<td>IGDSW ARMv8.5-MemTag</td>
</tr>
<tr>
<td>000</td>
<td>1010</td>
<td>010</td>
<td>CSW</td>
</tr>
<tr>
<td>000</td>
<td>1010</td>
<td>100</td>
<td>CGSW ARMv8.5-MemTag</td>
</tr>
<tr>
<td>000</td>
<td>1010</td>
<td>110</td>
<td>CGDSW ARMv8.5-MemTag</td>
</tr>
<tr>
<td>000</td>
<td>1110</td>
<td>010</td>
<td>CISW ARMv8.5-MemTag</td>
</tr>
<tr>
<td>000</td>
<td>1110</td>
<td>100</td>
<td>CIGSW ARMv8.5-MemTag</td>
</tr>
<tr>
<td>000</td>
<td>1110</td>
<td>110</td>
<td>CIGDSW ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>0100</td>
<td>001</td>
<td>ZVA ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>0100</td>
<td>011</td>
<td>GVA ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>0100</td>
<td>100</td>
<td>GZVA ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1010</td>
<td>001</td>
<td>CVAC ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1010</td>
<td>011</td>
<td>CGVAC ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1010</td>
<td>101</td>
<td>CGDVAC ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1011</td>
<td>001</td>
<td>CVAU ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1100</td>
<td>001</td>
<td>CVAP ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1100</td>
<td>011</td>
<td>CGVAP ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1100</td>
<td>101</td>
<td>CGDVAP ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1101</td>
<td>001</td>
<td>CVADP ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1101</td>
<td>011</td>
<td>CGVADP ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1110</td>
<td>001</td>
<td>CIVAC ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1110</td>
<td>011</td>
<td>CIGVAC ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>1110</td>
<td>101</td>
<td>CIGDVAC ARMv8.5-MemTag</td>
</tr>
</tbody>
</table>

\texttt{<op1>} is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

\texttt{<Cm>} is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

\texttt{<op2>} is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

\texttt{<Xt>} is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.
Operation

The description of SYS gives the operational pseudocode for this instruction.
DCPS1

Debug Change PE State to EL1, when executed in Debug state:

• If executed at EL0 changes the current Exception level and SP to EL1 using SP_EL1.
• Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS1 instruction is:

• EL1 if the instruction is executed at EL0.
• Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

• ELR_ELx becomes UNKNOWN.
• SPSR_ELx becomes UNKNOWN.
• ESR_ELx becomes UNKNOWN.
• DLR_EL0 and DSPSR_EL0 become UNKNOWN.
• The endianness is set according to SCTLR_ELx.EE.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and HCR_EL2.TGE == 1.
This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see DCPS.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 1 0 1 imm16 0 0 0 0 0 1 LL

System

DCPS1 {#<imm>}

bits(2) target_level = LL;
if LL == '00' then UNDEFINED;
if !Halted() then AArch64.UndefiendFault();

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

DCPSInstruction(target_level);

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DCPS2

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 if EL2 is disabled in the current Security state.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see DCPS.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 IMM16
```

System

DCPS2 {#<imm>}

\[
\text{bits(2) target\_level = LL;}
\text{if LL == '00' then UNDEFINED;}
\text{if !Halted() then AArch64.Undef\_inedFault();}
\]

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
DCPSInstruction(target\_level);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DCPS3

Debug Change PE State to EL3, when executed in Debug state:

• If executed at EL3 selects SP_EL3.
• Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

• ELR_EL3 becomes UNKNOWN.
• SPSR_EL3 becomes UNKNOWN.
• ESR_EL3 becomes UNKNOWN.
• DLRL_EL0 and DSRPR_EL0 become UNKNOWN.
• The endianness is set according to SCTLR_EL3.EE.

This instruction is UNDEFINED at all exception levels if either:

• ESDCR SDD == 1.
• EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see DCPS.

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

DCPSInstruction(target_level);

System

DCPS3 {#<imm>}

bits(2) target_level = LL;
if LL == '00' then UNDEFINED;
if !Halted() then AArch64.Undefindefault();
DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier.

System

DMB <option>|#<imm>

```plaintext
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if HaveSBExt() && CRm<3:0> == '0000' then
      op = MemBarrierOp_SB;
    else
      UNDEFINED;

case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
  otherwise
    if CRm<3:2> == '01' then
      op = MemBarrierOp_PSSBB;
    elsif CRm<3:2> == '00' && opc == '00' then
      op = MemBarrierOp_SSBB;
    elsif HaveSBExt() && CRm<3:2> == '00' && opc == '11' then
      op = MemBarrierOp_SB;
    else
      types = MBReqTypes_All;
      domain = MBReqDomain_FullSystem;
```

Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1110.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.
ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

<imm>
Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```plaintext
case op of
  when MemBarrierOp_DSB
    DataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMB
    DataMemoryBarrier(domain, types);
  when MemBarrierOp_ISB
    InstructionSynchronizationBarrier();
  when MemBarrierOp_SSB
    SpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBB
    SpeculativeSynchronizationBarrierToPA();
  when MemBarrierOp_SB
    SpeculationBarrier();
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DRPS

Debug restore process state.

System

```
if !Halted() || PSTATE.EL == EL0 then UNDEFINED;
```

Operation

```
DRPSInstruction();
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier*.

System

```
DSB <option>|#<imm>

MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
    when '00' op = MemBarrierOp_DSB;
    when '01' op = MemBarrierOp_DMB;
    when '10' op = MemBarrierOp_ISB;
    otherwise
        if HaveSBExt() && CRm<3:0> == '0000' then
            op = MemBarrierOp_SB;
        else
            UNDEFINED;
end case

case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
    otherwise
        if CRm<3:2> == '01' then
            op = MemBarrierOp_PSSBB;
        elsif CRm<3:2> == '00' && opc == '00' then
            op = MemBarrierOp_SSBB;
        elsif HaveSBExt() && CRm<3:2> == '00' && opc == '11' then
            op = MemBarrierOp_SB;
        else
            types = MBReqTypes_All;
            domain = MBReqDomain_FullSystem;
end case
```

Assembler Symbols

```
<option> Specifies the limitation on the barrier operation. Values are:

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.
```
ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

<imm>
Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```plaintext
case op of
  when MemBarrierOp_DSB
    DataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMB
    DataMemoryBarrier(domain, types);
  when MemBarrierOp_ISB
    InstructionSynchronizationBarrier();
  when MemBarrierOp_SSBB
    SpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBB
    SpeculativeSynchronizationBarrierToPA();
  when MemBarrierOp_SB
    SpeculationBarrier();
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DUP (element)

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (scalar).

It has encodings from 2 classes: Scalar and Vector

**Scalar**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------|--------------------------|--------------------------|
| 0 1 0 1 1 1 0 0 0 0 | imm5 | 0 0 0 0 0 1 | Rn | Rd |
```

**Scalar**

```
DUP <V><d>, <Vn>.<T>[<index>]
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;
```

**Vector**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------|--------------------------|--------------------------|
| 0 Q 0 0 1 1 1 0 0 0 0 | imm5 | 0 0 0 0 0 1 | Rn | Rd |
```

**Vector**

```
DUP <Vd>.<T>, <Vn>.<Ts>[<index>]
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

**Assembler Symbols**

<T> For the scalar variant: is the element width specifier, encoded in “imm5”:
For the vector variant: is an arrangement specifier, encoded in "imm5:Q":

<table>
<thead>
<tr>
<th>imm5</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>xxxx1</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>xxx10</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>xxx10</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>x1000</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Ts> Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<V> Is the destination width specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index> Is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
   Elem[result, e, esize] = element;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
DUP (general)

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

DUP <Vd><T>, <R><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;
// imm5<4:size+1> is IGNORED
if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<?Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<?T> Is an arrangement specifier, encoded in “imm5:Q”:

\[
\begin{array}{ccc}
\text{imm5} & \text{Q} & \text{T} \\
x\text{xxxx}0 & 0 & 8B \\
\text{xxxx}1 & 1 & 16B \\
x\text{xxx}0 & 0 & 4H \\
x\text{xxx}1 & 1 & 8H \\
x\text{xx}100 & 0 & 2S \\
x\text{xx}100 & 1 & 4S \\
x\text{x}1000 & 0 & \text{RESERVED} \\
x\text{x}1000 & 1 & 2D
\end{array}
\]

<?R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

\[
\begin{array}{c}
\text{imm5} \\
x\text{xxxx}0 & \text{RESERVED} \\
x\text{xxxx}1 & W \\
x\text{xxx}10 & W \\
x\text{xx}100 & W \\
x\text{x}1000 & X
\end{array}
\]

Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

<?n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bias(esize) element = X[n];
bias(datasize) result;

for e = 0 to elements-1
  \text{Elem}[\text{result}, e, esize] = element;
\text{V}[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
DVP

Data Value Prediction Restriction by Context prevents data value predictions, based on information gathered from earlier execution within an particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see *DVP RCTX, Data Value Prediction Restriction by Context*.

This is an alias of *SYS*. This means:

- The encodings in this description are named to match the encodings of *SYS*.
- The description of *SYS* gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & Rt \\
\end{array}
\]

**System**

\[
DVP\ RCTX,\ <Xt>
\]

is equivalent to

\[
SYS\ \#3,\ C8,\ C3,\ \#5,\ <Xt>
\]

and is always the preferred disassembly.

**Assembler Symbols**

\[
<Xt>
\]

Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

**Operation**

The description of *SYS* gives the operational pseudocode for this instruction.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

| sf | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| opc | 1  | 0  | 0  | 0  | 1  | 1  | Rm | imm6 | Rn | Rd |

32-bit (sf == 0)

EON <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

EON <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp AND; setflags = FALSE;
  when '01' op = LogicalOp ORR; setflags = FALSE;
  when '10' op = LogicalOp EOR; setflags = FALSE;
  when '11' op = LogicalOp AND; setflags = TRUE;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
Operation

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;
if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
EOR3

Three-way Exclusive OR performs a three-way exclusive OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

Advanced SIMD
(ARMv8.2)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 1 1 0 0 0 0 | Rm 0 | Ra | Rn | Rd
```

Advanced SIMD

EOR3 <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Va>` Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];
V[d] = Vn EOR Vm EOR Va;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
EOR (vector)

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

opc2

Three registers of the same type

EOR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bids(datasize) operand2;
bids(datasize) operand3;
bids(datasize) operand4 = V[n];

case op of
  when VBitOp_VEOR
    operand1 = V[m];
    operand2 = Zeros();
    operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
    operand2 = operand1;
    operand3 = NOT(V[m]);

V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**EOR (Immediate)**

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**32-bit (sf == 0 && N == 0)**

EOR <Wd|WSP>, <Wn>, #<imm>

**64-bit (sf == 1)**

EOR <Xd|SP>, <Xn>, #<imm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;

case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

**Assembler Symbols**

- <Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
EOR (shifted register)

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

EOR <Wd>, <Wn>, <Wm>{{, <shift> #<amount>}}</n>

64-bit (sf == 1)

EOR <Xd>, <Xn>, <Xm>{{, <shift> #<amount>}}</n>

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf == '1' \text{ then 64 else 32}; \\
\text{boolean } \text{setflags}; \\
\text{LogicalOp } \text{op}; \\
\text{case } \text{opc} \text{ of} \\
\text{when } '00' \text{ op } = \text{LogicalOp AND}; \text{ setflags } = \text{FALSE}; \\
\text{when } '01' \text{ op } = \text{LogicalOp ORR}; \text{ setflags } = \text{FALSE}; \\
\text{when } '10' \text{ op } = \text{LogicalOp EOR}; \text{ setflags } = \text{FALSE}; \\
\text{when } '11' \text{ op } = \text{LogicalOp AND}; \text{ setflags } = \text{TRUE}; \\
\text{if } sf == '0' \&\& \text{imm6<5> == '1'} \text{ then UNDEFINED}; \\
\text{ShiftType } \text{shift_type} = \text{DecodeShift(shift)}; \\
\text{integer } \text{shift_amount} = \text{UInt}(\text{imm6}); \\
\text{boolean } \text{invert} = (N == '1'); \\
\end{align*}
\]

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  - For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
Operation

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ERET**

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores PSTATE from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERET is UNDEFINED at EL0.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  |

**System**

\[
\text{if } \text{PSTATE.EL} == \text{EL0 } \text{then UNDEFINED;}
\]

boolean pac = (A == '1');
boolean use_key_a = (M == '0');

if !pac && op4 != '00000' then
UNDEFINED;
elsif pac && (!HavePACExt() || op4 != '11111') then
UNDEFINED;
if Rn != '11111' then
UNDEFINED;

**Operation**

\[
\text{AArch64.CheckForERetTrap}(\text{pac, use_key_a});
\]

bits(64) target = ELR[];
if pac then
  if use_key_a then
    target = AuthIA(ELR[], SP[]);
  else
    target = AuthIB(ELR[], SP[]);
AArch64.ExceptionReturn(target, SPSR[]);
ERETAA, ERETAB

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores \textit{PSTATE} from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for \texttt{ERETAA}, and key B is used for \texttt{ERETAB}.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated. The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See \textit{Illegal return events from AArch64 state}.

\texttt{ERETAA} and \texttt{ERETAB} are \texttt{UNDEFINED} at EL0.

### Integer (ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | M | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1 |

#### ERETAA (M == 0)

\texttt{ERETAA}

#### ERETAB (M == 1)

\texttt{ERETAB}

if \texttt{PSTATE.EL} == EL0 then UNDEFINED;

boolean pac = (A == '1');
boolean use_key_a = (M == '0');

if !pac && \texttt{op4} != '00000' then UNDEFINED;
elsif pac && (!\texttt{HavePACExt}() || \texttt{op4} != '11111') then UNDEFINED;
if \texttt{Rn} != '11111' then UNDEFINED;

\textbf{Operation}

\texttt{AArch64.CheckForERetTrap}(pac, use\_key\_a);
\texttt{bits(64) target = ELR[]};

if pac then
    if use\_key\_a then
        target = AuthIA(ELR[], SP[]);
    else
        target = AuthIB(ELR[], SP[]);
\texttt{AArch64.ExceptionReturn}(target, SPSR[]);
ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR_EL1 and VDISR_EL2.
This instruction can be used at all Exception levels and in Debug state.
In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the ARM(R) Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for ARMv8-A architecture profile.
If the RAS Extension is not implemented, this instruction executes as a NOP.

System
(ARMv8.2)

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

System

ESB

```c
SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111' // SEE "XPACLRI"
    when '0001 xxx' // SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716"
      when '0010 000'
        if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_ESB;
      when '0010 001'
        if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_PSB;
      when '0010 100'
        op = SystemHintOp_CSDB;
    when '0011 xxx' // SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP"
      when '0100 xx0'
        op = SystemHintOp_BTI;
        BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
    otherwise EndOfInstruction(); // Instruction executes as NOP
```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap(EL1, TRUE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap(EL3, TRUE);
      WaitForEvent();
  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap(EL1, FALSE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap(EL3, FALSE);
      WaitForInterrupt();
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    BTypeNext = '00';
  otherwise // do nothing
**EXT**

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows an example of the operation of EXT doubleword operation for $Q = 0$ and $imm4<2:0> = 3$.

![Example Figure]

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Advanced SIMD**

```plaintext
EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>
```

Integer $d = \text{UInt}(Rd)$;
Integer $n = \text{UInt}(Rn)$;
Integer $m = \text{UInt}(Rm)$;

If $Q == '0' \&\& imm4<3> == '1'$ then UNDEFINED;

Integer $\text{datasize} = \text{if } Q == '1' \text{ then 128 else 64}$;
Integer $\text{position} = \text{UInt}(imm4) << 3$;

**Assembler Symbols**

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<index>`: Is the lowest numbered byte element to be extracted, encoded in “Q:imm4”:

<table>
<thead>
<tr>
<th>Q</th>
<th>imm4&lt;3&gt;</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>imm4&lt;2:0&gt;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>imm4</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m];
bits(datasize) lo = V[n];
bits(datasize*2) concat = hi : lo;
V[d] = concat<position+datasize-1:position>;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**EXTR**

Extract register extracts a register from a pair of registers. This instruction is used by the alias **ROR (immediate)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>N</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imms</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0 && N == 0 && imms == 0xxxxx)

`EXTR <Wd>, <Wn>, <Wm>, #<lab>`

### 64-bit (sf == 1 & N == 1)

`EXTR <Xd>, <Xn>, <Xm>, #<lab>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
integer lsb;

if N != sf then UNDEFINED;
if sf == '0' && imms<5> == '1' then UNDEFINED;
lsb = UInt(imms);
```

### Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<lab>`: For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field.
  - For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROR (immediate)</strong></td>
<td>Rn == Rm</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(2*datasize) concat = operand1:operand2;
result = concat<lsb+datasize-1:lsb>;
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
FABD

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision (ARMv8.2)

```
0 1 1 1 1 1 0 1 1 0
```

Scalar half precision

```
FABD <Hd>, <Hn>, <Hm>
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

Scalar single-precision and double-precision

```
0 1 1 1 1 1 0 1 sz 1
```

Scalar single-precision and double-precision

```
FABD <V><d>, <V><n>, <V><m>
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

Vector half precision (ARMv8.2)

```
0 1 1 1 0 1 1 0
```

Vector half precision
Vector half precision

FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

Vector single-precision and double-precision

FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) diff;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FABS (vector)

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision (ARMv8.2)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | Rn | Rd |

**Half-precision**

```
FABS <Vd>.<T>, <Vn>.<T>
```  

```plaintext
if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

**Single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 1  | sz | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | Rn | Rd |

**Single-precision and double-precision**

```
FABS <Vd>.<T>, <Vn>.<T>
```  

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the “Rd” field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in “Q”:
  ```plaintext
  Q  <T>
  0  4H
  1  8H
  ```
  For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision (type == 11)
(ARMv8.2)

FABS <Hd>, <Hn>

Single-precision (type == 00)

FABS <Sd>, <Sn>

Double-precision (type == 01)

FABS <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11' if HaveFP16Ext() then
datasize = 16;
else
  UNDEFINED;
endcase;

FPUnaryOp fpop;

case opc of
  when '00' fpop = FPUnaryOp_MOV;
  when '01' fpop = FPUnaryOp_ABS;
  when '10' fpop = FPUnaryOp_NEG;
  when '11' fpop = FPUnaryOp_SQRT;
endcase;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
    when FPUntaryOp_MOV result = operand;
    when FPUntaryOp_ABS result = FPAbs(operand);
    when FPUntaryOp_NEG result = FPNeg(operand);
    when FPUntaryOp_SQRT result = FPSqrt(operand, FPCR);

V[d] = result;
```
FACGE

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|
| 0 1 1 1 1 1 0 0 1 0 | Rm | 0 0 1 0 1 1 | Rd |

U E ac

Scalar half precision

FACGE <Hd>, <Hn>, <Hm>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|
| 0 1 1 1 1 1 0 0 sz 1 | Rm | 1 1 1 0 1 1 | Rd |

U E ac
Scalar single-precision and double-precision

\( \text{FACGE} \langle \mathbf{V} \rangle <\mathbf{d}, <\mathbf{V} \rangle <\mathbf{n}, <\mathbf{V} \rangle <\mathbf{m} \)

integer \( d = \text{UInt}(\text{Rd}) \);
integer \( n = \text{UInt}(\text{Rn}) \);
integer \( m = \text{UInt}(\text{Rm}) \);
integer \( \text{esize} = 32 << \text{UInt}(\text{sz}) \);
integer \( \text{datasize} = \text{esize} \);
integer \( \text{elements} = 1 \);
CompareOp \( \text{cmp} \);
boolean \( \text{abs} \);

\text{case E:U:a of}
\begin{align*}
\text{when '000'} & \text{ cmp = CompareOp EQ; abs = FALSE; } \\
\text{when '010'} & \text{ cmp = CompareOp GE; abs = FALSE; } \\
\text{when '011'} & \text{ cmp = CompareOp GE; abs = TRUE; } \\
\text{when '110'} & \text{ cmp = CompareOp GT; abs = FALSE; } \\
\text{when '111'} & \text{ cmp = Compareop GT; abs = TRUE; } \\
\text{otherwise} & \text{ UNDEFINED; }
\end{align*}

Vector half precision

(ARMv8.2)

\[ \begin{array}{c|cccc|c|cccc|c}
\hline
\hline
\text{U} & \text{Q} & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & \text{Rm} & 0 & 0 & 1 & 0 & 1 & 1 & \text{Rn} & \text{Rd} & \text{ac} \\
\hline
\end{array} \]

Vector half precision

\( \text{FACGE} \langle \mathbf{Vd} \rangle .<\mathbf{T}>, <\mathbf{Vn} \rangle .<\mathbf{T}>, <\mathbf{Vm} \rangle .<\mathbf{T} > \)

if !\( \text{HaveFP16Ext}() \) then UNDEFINED;

integer \( d = \text{UInt}(\text{Rd}) \);
integer \( n = \text{UInt}(\text{Rn}) \);
integer \( m = \text{UInt}(\text{Rm}) \);
integer \( \text{esize} = 16 \);
integer \( \text{datasize} = \text{if Q == '1' then 128 else 64} \);
integer \( \text{elements} = \text{datasize DIV esize} \);
CompareOp \( \text{cmp} \);
boolean \( \text{abs} \);

\text{case E:U:ac of}
\begin{align*}
\text{when '000'} & \text{ cmp = CompareOp EQ; abs = FALSE; } \\
\text{when '010'} & \text{ cmp = CompareOp GE; abs = FALSE; } \\
\text{when '011'} & \text{ cmp = CompareOp GE; abs = TRUE; } \\
\text{when '110'} & \text{ cmp = Compareop GT; abs = FALSE; } \\
\text{when '111'} & \text{ cmp = CompareOp GT; abs = TRUE; } \\
\text{otherwise} & \text{ UNDEFINED; }
\end{align*}

Vector single-precision and double-precision

\[ \begin{array}{c|cccc|c|cccc|c}
\hline
\hline
\text{U} & \text{Q} & 1 & 0 & 1 & 1 & 1 & 0 & 0 & \text{sz} & 1 & \text{Rm} & 1 & 1 & 1 & 0 & 1 & 1 & \text{Rn} & \text{Rd} & \text{ac} \\
\hline
\end{array} \]
Vector single-precision and double-precision

FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

FACGT

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |

U    E   ac

Scalar half precision

FACGT <Hd>, <Hn>, <Hm>
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = Compareop_GT; abs = TRUE;
  otherwise UNDEFINED;

Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | sz|1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |

U    E   ac
Scalar single-precision and double-precision

\texttt{FACGT \langle V \rangle <d>, \langle V \rangle <n>, \langle V \rangle <m>}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

\textbf{case} E:U:ac \textbf{of}
  \textbf{when} '000' cmp = CompareOp_EQ; abs = FALSE;
  \textbf{when} '010' cmp = CompareOp_GE; abs = FALSE;
  \textbf{when} '011' cmp = CompareOp_GE; abs = TRUE;
  \textbf{when} '110' cmp = CompareOp_GT; abs = FALSE;
  \textbf{when} '111' cmp = CompareOp_GT; abs = TRUE;
\textbf{otherwise} UNDEFINED;
\end{verbatim}

Vector half precision

(\textit{ARMv8.2})

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

\textbf{case} E:U:ac \textbf{of}
  \textbf{when} '000' cmp = CompareOp_EQ; abs = FALSE;
  \textbf{when} '010' cmp = CompareOp_GE; abs = FALSE;
  \textbf{when} '011' cmp = CompareOp_GE; abs = TRUE;
  \textbf{when} '110' cmp = CompareOp_GT; abs = FALSE;
  \textbf{when} '111' cmp = CompareOp_GT; abs = TRUE;
\textbf{otherwise} UNDEFINED;
\end{verbatim}

Vector single-precision and double-precision
Vector single-precision and double-precision

```
FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```plaintext
d = Uint(Rd);
n = Uint(Rn);
m = Uint(Rm);
if sz:Q == '10' then UNDEFINED;
esize = 32 << Uint(sz);
datasize = if Q == '1' then 128 else 64;
elements = datasize DIV esize;
cmp = CompareOp选址;
abs = boolean;
```
```
```case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Assembler Symbols

- `<Hd>` is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<V>` is a width specifier, encoded in "sz":
<table>
<thead>
<tr>
<th>sz</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
- `<d>` is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` for the vector half precision variant: is an arrangement specifier, encoded in "Q":
<table>
<thead>
<tr>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
  For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```
FADD (vector)

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|------------------|
| 0               | Q   | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 0  | 1  | 0  | 1  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   | U |
```

Half-precision

```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Single-precision and double-precision

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|------------------|
| 0               | Q   | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 1  | 0  | 1  | Rn |   |   |   |   |   |   |   |   |   |   |   |   | U |
```

Single-precision and double-precision

```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Assembler Symbols

```
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
```

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”: 

FADD (vector)
\[ \begin{array}{c|cc}
\text{sz} & \text{Q} & <T> \\
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & \text{RESERVED} \\
1 & 1 & 2D \\
\end{array} \]

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPAdd(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | type | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 0  | 0  | 0  | 1  | 0  | 1  | 0  | Rd   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Half-precision (type == 11)
(ARMv8.2)

FADD <Hd>, <Hn>, <Hm>

Single-precision (type == 00)

FADD <Sd>, <Sn>, <Sm>

Double-precision (type == 01)

FADD <Dd>, <Dn>, <Dm>

```plaintext
type = "00" datasize = 32;
type = "01" datasize = 64;
type = "10" UNDEFINED;
type = "11"
    if HaveFP16Ext() then
dataize = 16;
    else
UNDEFINED;

boolean sub_op = (op == '1');</plaintext>

Assembler Symbols

- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Dn>** Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Hd>** Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Hn>** Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Hm>** Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Sd>** Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Sn>** Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Sm>** Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if sub_op then
    result = FPSub(operand1, operand2, FPCR);
else
    result = FPAdd(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rn | Rn | Rd |

sz

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rn | Rn | Rd |

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rn | Rn | Rd |

Assembler Symbols

For the half-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RESERVED</td>
<td></td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:
\( sz <V> \)

<table>
<thead>
<tr>
<th>sz</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\) For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

\[
\text{CheckFPAdvSIMDEnabled64}(); \\
\text{bits}(\text{datasize}) \text{ operand} = V[n]; \\
V[d] = \text{Reduce}(\text{op}, \text{operand}, \text{esize});
\]

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00beta8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FADDP (vector)

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 0  | 1  | 0  | 1  | Rn | Rd |

Half-precision

FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 1  | 0  | 1  | Rn | Rd |

Single-precision and double-precision

FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”: .
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d] = result;
```

---

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Three registers of the same type

<table>
<thead>
<tr>
<th>(ARMv8.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

### Three registers of the same type

FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

```plaintext
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

- **<Vd>**
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<T>**
  - Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- **<Vn>**
  - Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- **<Vm>**
  - Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

- **<rotate>**
  - Is the rotation, encoded in “rot”:

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>1</td>
<td>270</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2) -1
  case rot of
    when '0'
      element1 = FPNeg(Elem[operand2, e*2+1, esize]);
      element3 = Elem[operand2, e*2, esize];
    when '1'
      element1 = Elem[operand2, e*2+1, esize];
      element3 = FPNeg(Elem[operand2, e*2, esize]);
  Elem[result, e*2, esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR);
  Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR);

V[d] = result;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the \textit{PSTATE}. \{N, Z, C, V\} flags. If the condition does not pass then the \textit{PSTATE}. \{N, Z, C, V\} flags are set to the flag bit specifier.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPSCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 0 type 1 Rm  cond 0 1 Rn 0 nzcv
\end{verbatim}

\textbf{Half-precision (type == 11)}

(ARMv8.2)

\begin{verbatim}
FCCMP <Hn>, <Hm>, #<nzcv>, <cond>
\end{verbatim}

\textbf{Single-precision (type == 00)}

\begin{verbatim}
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
\end{verbatim}

\textbf{Double-precision (type == 01)}

\begin{verbatim}
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
\end{verbatim}

integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
   when '00' datasize = 32;
   when '01' datasize = 64;
   when '10' UNDEFINED;
   when '11'
      if HaveFP16Ext() then
datasize = 16;
else
   UNDEFINED;

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;

\textbf{Assembler Symbols}

\begin{verbatim}
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

NaNs
The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.

**Operation**

```c
CheckFPAvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];

if ConditionHolds(condition) then
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCCMPE

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE \{N, Z, C, V\} flags. If the condition does not pass then the PSTATE \{N, Z, C, V\} flags are set to the flag bit specifier. If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | type | 1  | Rm | cond | 0 | 1 | Rn | 1 | nzcv |
```

**Half-precision (type == 11)**

(ARMv8.2)

FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>

**Single-precision (type == 00)**

FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>

**Double-precision (type == 01)**

FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>

```plaintext
type
Rm
cond
Rn
nzcv
```

integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
datasize = 16;
    else
      UNDEFINED;
    end

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<nzcv>` Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
- `<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

NaNs

**FCCMPE**  
Page 256
The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.

FCCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation

```c
CheckFPAAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];
if ConditionHolds(condition) then
  flags = FPCompare(operand1, operand2, signal_all_nans, FPCR);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00beta8_rc3; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMEQ (register)

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(ARMv8.2)

```
0 1 0 1 1 1 0 0 1 0 Rm 0 0 1 0 0 1 Rn Rd
U E ac
```

Scalar single-precision and double-precision

```
0 1 0 1 1 1 0 0 sz 1 Rm 1 1 1 0 0 1 Rn Rd
U E ac
```
Scalar single-precision and double-precision

FCMEQ <V><d>, <V><n>, <V><m>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;
```

Vector half precision
(ARMv8.2)

```plaintext
|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 0  | 1  | Rn |  | Rd |
| U  | E  | ac |

Vector half precision

FCMEQ <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;
```

Vector single-precision and double-precision

```plaintext
|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 0  | 1  | Rn |  | Rd |
| U  | E  | ac |
```
**Vector single-precision and double-precision**

FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer e;size = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV e;size;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

**Assembler Symbols**

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
CheckFPAdvSIMEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  if abs then
    element1 = FPAbs(element1);
    element2 = FPAbs(element2);
  case cmp of
    when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
    when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
    when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
FCMEQ (zero)

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision.

Scalar half precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  |

Rn  Rd

U

op

Scalar half precision

FCMEQ <Hd>, <Hn>, #0.0

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;

case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;

Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | sz| 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  |

Rn  Rd

U

op

Scalar single-precision and double-precision

FCMEQ <V><d>, <V><n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;

case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
Vector half precision
(ARMv8.2)

```
0 Q 0 0 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 0
U

op

Rn  Rd
```

Vector half precision

FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
  case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

```
0 Q 0 0 1 1 1 0 | sz | 1 0 0 0 0 0 0 1 1 0 1 1 0
U

op

Rn  Rd
```

Vector single-precision and double-precision

FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
  case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;

Assembler Symbols

<

<Hd>
  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn>
  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V>
  Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d>
  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n>
  Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
    when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMGE (register)

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 0  | 1  | Rd | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | ac |
| U  | E  | ac |

Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 0  | 0  | 1  | Rd | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | ac |

FCMGE <Hd>, <Hn>, <Hm>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp EQ; abs = FALSE;
  when '010' cmp = CompareOp GE; abs = FALSE;
  when '011' cmp = CompareOp GE; abs = TRUE;
  when '110' cmp = CompareOp GT; abs = FALSE;
  when '111' cmp = CompareOp GT; abs = TRUE;
  otherwise UNDEFINED;
Scalar single-precision and double-precision

FCMGE \(<V><d>, <V><n>, <V><m>

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer esize = 32 << Uint(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector half precision
(ARMv8.2)

Vector half precision

FCMGE \(<Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>E</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>E</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector single-precision and double-precision

FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ  test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE  test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT  test_passed = FPCompareGT(element1, element2, FPCR);
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```
FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPSCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

<p>| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

U   op   Rd   Rn

Scalar half precision

FCMGE <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;

Scalar single-precision and double-precision

|      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0    | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |

U   op   Rd   Rn

Scalar single-precision and double-precision

FCMGE <V<d>, <V<n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
Vector half precision

(ARMv8.2)

**FCMGE** \(<V_d>.<T>, <V_n>.<T>, #0.0\)

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**CompareOp** comparison;
case op:U of
  when '00' comparison = **CompareOp_GT**;
  when '01' comparison = **CompareOp_GE**;
  when '10' comparison = **CompareOp_EQ**;
  when '11' comparison = **CompareOp_LE**;

Vector single-precision and double-precision

**FCMGE** \(<V_d>.<T>, <V_n>.<T>, #0.0\)

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**CompareOp** comparison;
case op:U of
  when '00' comparison = **CompareOp_GT**;
  when '01' comparison = **CompareOp_GE**;
  when '10' comparison = **CompareOp_EQ**;
  when '11' comparison = **CompareOp_LE**;

**Assembler Symbols**

\(<H_d>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<H_n>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

\(<V>\) Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
    when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMGT (register)

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 0  | 1  | Rn | Rd |
| U  | E  | ac |

Scalar single-precision and double-precision

FCMGT <Hd>, <Hn>, <Hm>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
when '000' cmp = CompareOp_EQ; abs = FALSE;
when '010' cmp = CompareOp_GE; abs = FALSE;
when '011' cmp = CompareOp_GE; abs = TRUE;
when '110' cmp = CompareOp_GT; abs = FALSE;
when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 0  | 1  | Rn | Rd |
| U  | E  | ac |
Scalar single-precision and double-precision

FCMGT <V<d>, <V<n>, <V<m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector half precision
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 0  | 1  | Rn | 1  | 1  | 1  | 0  | 0  | 1  | Rd |

U E ac

Vector half precision

FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | 1  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 0  | 1  | Rn | 1  | 1  | 1  | 0  | 0  | 1  | Rd |

U E ac

FCMGT (register)
Vector single-precision and double-precision

FCMGT $<\text{Vd}>., <\text{T}>, <\text{Vn}>., <\text{Vm}>.<$

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;
```

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise  UNDEFINED;

Assembler Symbols

$<\text{Hd}>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

$<\text{Hn}>$ Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

$<\text{Hm}>$ Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

$<\text{V}>$ Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>$&lt;\text{V}&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

$<\text{d}>$ Is the number of the SIMD&FP destination register, in the "Rd" field.

$<\text{n}>$ Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

$<\text{m}>$ Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

$<\text{Vd}>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<\text{T}>$ For the vector half precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>$&lt;\text{T}&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>$&lt;\text{T}&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<\text{Vn}>$ Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$<\text{Vm}>$ Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
\end{verbatim}

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMGT (zero)

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U  U  U

op

type

Scalar half precision

```
FCMGT <Hd>, <Hn>, #0.0

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U  U  U

op

type

Scalar single-precision and double-precision

```
FCMGT <V><d>, <V><n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```
Vector half precision

(ARMv8.2)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 0 0 1 1 1 0 1 1 1 1 0 0 0 1 1 0 0 1 0 1

Rn | Rd
---|---
```

**Vector half precision**

\[
\text{FCMGT} \ <V_d>.<T>, \ <V_n>.<T>, \ #0.0
\]

if \(!\text{HaveFP16Ext}()\) then UNDEFINED;

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

integer \(\text{esize} = 16\);
integer \(\text{datasize} = \text{if } Q = '1' \text{ then } 128 \text{ else } 64\);
integer \(\text{elements} = \text{datasize} \div \text{esize}\);

```java
CompareOp \ comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 0 0 1 1 1 0 1 sz 1 0 0 0 0 0 1 1 0 0 1 0 1

Rn | Rd
---|---
```

**Vector single-precision and double-precision**

\[
\text{FCMGT} \ <V_d>.<T>, \ <V_n>.<T>, \ #0.0
\]

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

if \(sz:Q = '10'\) then UNDEFINED;
integer \(\text{esize} = 32 << \text{UInt}(sz)\);
integer \(\text{datasize} = \text{if } Q = '1' \text{ then } 128 \text{ else } 64\);
integer \(\text{elements} = \text{datasize} \div \text{esize}\);

```java
CompareOp \ comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

**Assembler Symbols**

\(<Hd>\)
Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Hn>\)
Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

\(<V>\)
Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\)
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<n>\)
Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
For the vector half precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```cpp
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);

    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMLA (by element)

Floating-point Complex Multiply Accumulate (by element).
This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Vector**
*(ARMv8.3)*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>size</th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th>0</th>
<th>rot</th>
<th>1</th>
<th>H</th>
<th>0</th>
<th>Rd</th>
</tr>
</thead>
</table>

**(size == 01)**

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

**(size == 10)**

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
if size == '00' || size == '11' then UNDEFINED;
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integerdatasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q=='0' then UNDEFINED;
```

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in "size:H:L":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<rotate> Is the rotation, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>180</td>
</tr>
<tr>
<td>11</td>
<td>270</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bbits(datasize) operand1 = V[n];
bbits(datasize) operand2 = V[m];
bbits(datasize) operand3 = V[d];
bbits(datasize) result;
for e = 0 to (elements DIV 2) -1
  case rot of
    when '00'
      element1 = Elem[operand2, index*2, esize];
      element2 = Elem[operand1, e*2, esize];
      element3 = Elem[operand2, index*2+1, esize];
      element4 = Elem[operand1, e*2, esize];
    when '01'
      element1 = FPNeg(Elem[operand2, index*2+1, esize]);
      element2 = Elem[operand1, e*2+1, esize];
      element3 = Elem[operand2, index*2, esize];
      element4 = Elem[operand1, e*2, esize];
    when '10'
      element1 = FPNeg(Elem[operand2, index*2+1, esize]);
      element2 = Elem[operand1, e*2, esize];
      element3 = FPNeg(Elem[operand2, index*2, esize]);
      element4 = Elem[operand1, e*2+1, esize];
    when '11'
      element1 = Elem[operand2, index*2+1, esize];
      element2 = Elem[operand1, e*2+1, esize];
      element3 = FPNeg(Elem[operand2, index*2, esize]);
      element4 = Elem[operand1, e*2, esize];
  Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
  Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCMLA**

Floating-point Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Three registers of the same type

(ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | 0  | Rm | 1  | 1  | 0  | rot | 1  | Rn | 1  | Rd |

### Three registers of the same type

FCMLA <Vd>..<T>, <Vn>..<T>, <Vm>..<T>, #<rotate>

```plaintext
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

- **<Vd>**
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>**
  - Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- **<Vn>**
  - Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>**
  - Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<rotate>**
  - Is the rotation, encoded in “rot”:
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;

for e = 0 to (elements DIV 2) -1
    case rot of
        when '00'
            element1 = Elem[operand2, e*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, e*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, e*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, e*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];
    Elem[result, e*2,   esize] = FPMulAdd(Elem[operand3, e*2,   esize], element2, element1, FPCR);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMLE (zero)

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

**Scalar half precision**

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U op

**Scalar single-precision and double-precision**

```
FCMLE <V>d, <V>n, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
\- when '00' comparison = CompareOp_GT;
\- when '01' comparison = CompareOp_GE;
\- when '10' comparison = CompareOp_EQ;
\- when '11' comparison = CompareOp_LE;
```

**Scalar single-precision and double-precision**

```
FCMLE <V>d, <V>n, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
\- when '00' comparison = CompareOp_GT;
\- when '01' comparison = CompareOp_GE;
\- when '10' comparison = CompareOp_EQ;
\- when '11' comparison = CompareOp_LE;
```
Vector half precision

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U

op

Vector half precision

FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);  
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);    
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);    
    when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);    
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);    
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00beta8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCMLT (zero)**

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

### Scalar half precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|
| 0 1 1 1 1 0 1 1 1 1 0 0 0 1 1 1 0 1 0 | Rn  Rd                           |

**FCMLT** `<Hd>, <Hn>, #0.0`

If `!HaveFP16Ext()` then UNDEFINED;

integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;

integer esize = 16;
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison = **CompareOp_LT**;

### Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|
| 0 1 1 1 1 0 1 sz 0 0 0 0 0 1 1 1 0 1 0 | Rn  Rd                           |

**FCMLT** `<V><d>, <V><n>, #0.0`

integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;

integer esize = 32 `<<` `UInt(sz)`;
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison = **CompareOp_LT**;

### Vector half precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|
| 0 Q 0 1 1 1 0 1 1 1 1 0 0 0 1 1 1 0 1 0 | Rn  Rd                           |
Vector half precision

FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;

Vector single-precision and double-precision

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMP

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 1 0 type 1 Rm 0 0 1 0 0 0 Rn 0 x 0 0 0
opc

Half-precision (type == 11 && opc == 00)
(ARMv8.2)

FCMP <Hn>, <Hm>

Half-precision, zero (type == 11 && Rm == (00000) && opc == 01)
(ARMv8.2)

FCMP <Hn>, #0.0

Single-precision (type == 00 && opc == 00)

FCMP <Sn>, <Sm>

Single-precision, zero (type == 00 && Rm == (00000) && opc == 01)

FCMP <Sn>, #0.0

Double-precision (type == 01 && opc == 00)

FCMP <Dn>, <Dm>

Double-precision, zero (type == 01 && Rm == (00000) && opc == 01)

FCMP <Dn>, #0.0

integer n = UInt(Rn);
integer m = UInt(Rm);  // ignored when opc<0> == '1'

integer datasize;

\[
\text{case } \text{type of}
\begin{align*}
\text{when '00' } & \text{ datasize = 32}; \\
\text{when '01' } & \text{ datasize = 64}; \\
\text{when '10' } & \text{ UNDEFINED}; \\
\text{when '11' } & \\
\text{if HaveFP16Ext() then} & \text{ datasize = 16}; \\
\text{else} & \text{ UNDEFINED};
\end{align*}
\]

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');

Assembler Symbols

<\text{Dn}>  For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\textit{Dm}>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<\textit{Nn}>

For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\textit{Nm}>

Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<\textit{Sn}>

For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\textit{Sm}>

Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of \(<\), \(\leq\), \(>\) or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the \textit{FPSCR} flags being set to \(N=0\), \(Z=0\), \(C=1\), and \(V=1\).

\textbf{Operation}

\begin{verbatim}
CheckFPAdvSIMDEnabled64();

\begin{verbatim}
bis(datasize) operand1 = V[n];
\end{verbatim}
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
\end{verbatim}

\end{verbatim}

Internal version only: isa v3.25, AdvSIMD v27.01, pseudocode v85-xml-00beta8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMPE

Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the \textit{PSTATE}. \{N, Z, C, V\} flags.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Instruction Encoding](image)

**Half-precision (type == 11 && opc == 10)**

(ARMv8.2)

\[
\text{FCMPE} <\text{Hn}>, <\text{Hm}>
\]

**Half-precision, zero (type == 11 && Rm == (00000) && opc == 11)**

(ARMv8.2)

\[
\text{FCMPE} <\text{Hn}>, #0.0
\]

**Single-precision (type == 00 && opc == 10)**

\[
\text{FCMPE} <\text{Sn}>, <\text{Sm}>
\]

**Single-precision, zero (type == 00 && Rm == (00000) && opc == 11)**

\[
\text{FCMPE} <\text{Sn}>, #0.0
\]

**Double-precision (type == 01 && opc == 10)**

\[
\text{FCMPE} <\text{Dn}>, <\text{Dm}>
\]

**Double-precision, zero (type == 01 && Rm == (00000) && opc == 11)**

\[
\text{FCMPE} <\text{Dn}>, #0.0
\]

**Assembler Symbols**

- \texttt{<Dn>} For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

integer \texttt{n} = \texttt{UInt}(Rn);
integer \texttt{m} = \texttt{UInt}(Rm); // ignored when \texttt{opc<0> == '1'}

integer \texttt{datasize};

\textbf{case} type of
  \textbf{when} '00' \texttt{datasize} = 32;
  \textbf{when} '01' \texttt{datasize} = 64;
  \textbf{when} '10' \texttt{UNDEFINED};
  \textbf{when} '11'
    \textbf{if} \texttt{HaveFP16Ext}() \textbf{then}
      \texttt{datasize} = 16;
    \textbf{else}
      \texttt{UNDEFINED};

\texttt{boolean signal\_all\_nans} = (\texttt{opc<1> == '1'});
\texttt{boolean cmp\_with\_zero} = (\texttt{opc<0> == '1'});

Page 291
For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\m>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Hn>
For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hm>
Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sn>
For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sm>
Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

NaNs
The IEEE 754 standard specifies that the result of a comparison is precisely one of <, =, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.

FCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = if cmp_with_zero then FPZero('0') else V[m];
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```

FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
| 0  | 0 | 1 | 1 | 1 | 0 | type | 1 | Rm | cond | 1 | 1 | Rn | Rd |

Half-precision (type == 11)
(ARMv8.2)

FCSEL <Hd>, <Hn>, <Hm>, <cond>

Single-precision (type == 00)

FCSEL <Sd>, <Sn>, <Sm>, <cond>

Double-precision (type == 01)

FCSEL <Dd>, <Dn>, <Dm>, <cond>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
data size = 16;
    else
      UNDEFINED;
  bits(4) condition = cond;

Assembler Symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<DD> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<DD> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<DD> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<DD> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<DD> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<DD> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<DD> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<DD> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.
Operation

```c
CheckFPAdvSIMEnabled64();
bits(datasize) result;

result = if ConditionHolds(condition) then V[n] else V[m];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the FPCR and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision to single-precision (type == 11 && opc == 00)

\[
\text{FCVT} \ <Sd>, <Hn>
\]

### Half-precision to double-precision (type == 11 && opc == 01)

\[
\text{FCVT} \ <Dd>, <Hn>
\]

### Single-precision to half-precision (type == 00 && opc == 11)

\[
\text{FCVT} \ <Hd>, <Sn>
\]

### Single-precision to double-precision (type == 00 && opc == 01)

\[
\text{FCVT} \ <Dd>, <Sn>
\]

### Double-precision to half-precision (type == 01 && opc == 11)

\[
\text{FCVT} \ <Hd>, <Dn>
\]

### Double-precision to single-precision (type == 01 && opc == 00)

\[
\text{FCVT} \ <Sd>, <Dn>
\]

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if type == opc then UNDEFINED;

type srcsize;
case type of
    when '00' srcsize = 32;
    when '01' srcsize = 64;
    when '10' UNDEFINED;
    when '11' srcsize = 16;
type dstsize;
case opc of
    when '00' dstsize = 32;
    when '01' dstsize = 64;
    when '10' UNDEFINED;
    when '11' dstsize = 16;
```

### Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(dstsize) result;
bits(srcsize) operand = V[n];

result = FPConvert (operand, FPCR);
V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTAS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision (ARMv8.2)

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
|   0   | 1   | 0   | 1   | 1   | 1   | 0   | 0   | 1   | 1   | 1   | 0   | 0   | 1   | 0   | Rn   |
U   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
  Rd
```

Scalar single-precision and double-precision

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
|   0   | 1   | 0   | 1   | 1   | 1   | 0   | 0   | sz | 1   | 0   | 0   | 0   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | Rn   |
U   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
  Rd
```

Scalar single-precision and double-precision

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasetsize = esize;
integer elements = 1;
FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector half precision (ARMv8.2)

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
|   0   | 1   | 0   | 1   | 1   | 1   | 0   | 0   | 0   | 1   | 1   | 1   | 0   | 0   | 1   | 0   | Rn   |
U   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
  Rd
```

Vector half precision

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);
integer datasetsize = esize;
integer elements = 1;
FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```
Vector half precision

FCVTAS \(<Vd>\).<T>, \(<Vn>\).<T>

if !HaveFP16Ext() then UNDEFINED;

integer \(d\) = UInt(Rd);
integer \(n\) = UInt(Rn);

integer esize = 16;
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

integer \(d\) = UInt(Rd);
integer \(n\) = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Assembler Symbols

\(<Hd>\), Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<Hn>\), Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\(<V>\), Is a width specifier, encoded in "sz":

\[
\begin{array}{c|c}
\text{sz} & \text{<V>} \\
\hline
0 & S \\
1 & D \\
\end{array}
\]

\(<d>\), Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\(<n>\), Is the number of the SIMD&FP source register, encoded in the "Rn" field.
\(<Vd>\), Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\), For the vector half precision variant: is an arrangement specifier, encoded in "Q":

\[
\begin{array}{c|c|c}
\text{Q} & \text{<T>} \\
\hline
0 & \text{4H} \\
1 & \text{8H} \\
\end{array}
\]

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\[
\begin{array}{c|c|c|c}
\text{sz} & \text{Q} & \text{<T>} \\
\hline
0 & 0 & \text{2S} \\
0 & 1 & \text{4S} \\
1 & 0 & \text{RESERVED} \\
1 & 1 & \text{2D} \\
\end{array}
\]

\(<Vn>\), Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \text{V}[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
   element = \text{Elem}[\text{operand}, e, \text{esize}];
   \text{Elem}[\text{result}, e, \text{esize}] = \text{FPToFixed}(element, 0, \text{unsigned}, \text{FPCR}, \text{rounding});
\text{V}[d] = \text{result};
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| sf | 0 | 0 | 1 | 1 | 1 | 0 | type | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
| rmode | opcode |
Half-precision to 32-bit (sf == 0 && type == 11) (ARMv8.2)

FCVTAS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11) (ARMv8.2)

FCVTAS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && type == 00)

FCVTAS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 00)

FCVTAS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 01)

FCVTAS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 01)

FCVTAS <Xd>, <Dn>
integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  otherwise
    UNDEFINED;

case opcode<2:1>:rmode of
  when '00' xx'        // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00'        // [U]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00'        // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00'        // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01'        // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11'        // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UNDEFINED;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFDP(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        fltval = Vpart[n,part];
        intval = ZeroExtend(fltval, intsize);
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        fltval = intval<fltsize-1:0>;
        Vpart[d,part] = fltval;
    when FPConvOp_CVT_FtoI_JS
        fltval = V[n];
        intval = FPToFDPJS(fltval, FPCR, TRUE);
        X[d] = ZeroExtend(intval<31:0>, 64);
```
FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in \( \text{FPCR} \), the exception results in either a flag being set in \( \text{FPSR} \), or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the \( \text{CPACR_EL1}, \text{CPTR_EL2}, \) and \( \text{CPTR_EL3} \) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rd |
| U  |

#### Scalar half precision

\`
FCVTAU <Hd>, <Hn>
```

if \( \text{!HaveFP16Ext}() \) then UNDEFINED;

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

\text{FPRounding} \text{rounding} = \text{FPRounding\_TIEAWAY};

boolean unsigned = (U == '1');

### Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rd |
| U  |

#### Scalar single-precision and double-precision

\`
FCVTAU <V><d>, <V><n>
```

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);

integer esize = 32 \times \text{UInt}(sz);
integer datasize = esize;
integer elements = 1;

\text{FPRounding} \text{rounding} = \text{FPRounding\_TIEAWAY};

boolean unsigned = (U == '1');

### Vector half precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rd |
| U  |
Vector half precision

FCVTAU \(<V_d>.<T>, <V_n>.<T>\)

if !\texttt{HaveFP16Ext()} then UNDEFINED;

integer \(d = \texttt{UInt}(R_d);\)
integer \(n = \texttt{UInt}(R_n);\)

integer esize = 16;
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

\texttt{FPRounding} rounding = \texttt{FPRounding\_TIEAWAY};
boolean unsigned = (U == '1');

Vector single-precision and double-precision

\[
\begin{array}{ccccccccccccccc}
\hline
0 & 1 & 0 & 1 & 1 & 0 & 0 & sz & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & \texttt{Rn} & \texttt{Rd} \\
U &
\end{array}
\]

Vector single-precision and double-precision

FCVTAU \(<V_d>.<T>, <V_n>.<T>\)

integer \(d = \texttt{UInt}(R_d);\)
integer \(n = \texttt{UInt}(R_n);\)

if \(sz:Q == '10'\) then UNDEFINED;
integer esize = 32 \(<\texttt{UInt}(sz);\)
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

\texttt{FPRounding} rounding = \texttt{FPRounding\_TIEAWAY};
boolean unsigned = (U == '1');

Assembler Symbols

\(<H_d>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<H_n>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\(<V>\) Is a width specifier, encoded in "sz":

\[
\begin{array}{c|c}
sz & \texttt{<V>} \\
0 & S \\
1 & D \\
\end{array}
\]

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\(<n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.
\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

\[
\begin{array}{c|c}
Q & \texttt{<T>} \\
0 & \texttt{4H} \\
1 & \texttt{8H} \\
\end{array}
\]

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

\[
\begin{array}{c|c|c}
sz & Q & \texttt{<T>} \\
0 & 0 & \texttt{2S} \\
0 & 1 & \texttt{4S} \\
1 & 0 & \texttt{RESERVED} \\
1 & 1 & \texttt{2D} \\
\end{array}
\]

\(<V_n>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPtoFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| sf | 0 | 0 | 1 | 1 | 1 | 0 | type | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
| rmode | opcode |
Half-precision to 32-bit (sf == 0 && type == 11)  
(ARMv8.2)

  FCVTAU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11)  
(ARMv8.2)

  FCVTAU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && type == 00)

  FCVTAU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 00)

  FCVTAU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 01)

  FCVTAU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 01)

  FCVTAU <Xd>, <Dn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
when '00'
  fltsize = 32;
when '01'
  fltsize = 64;
when '10'
  if opcode<2:1>:rmode != '11 01' then UNDEFINED;
  fltsize = 128;
when '11'
  if HaveFP16Ext() then
    fltsize = 16;
  else
    UNDEFINED;
end case;

case opcode<2:1>:rmode of
when '00 xx'  // FCVT[NPMZ][US]
  rounding = FPDecodeRounding(rmode);
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_FtoI;
when '01 00'  // [US]CVTF
  rounding = FPRoundingMode(FPCR);
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_ItoF;
when '10 00'  // FCVTA[US]
  rounding = FPRounding_TIEAWAY;
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_FtoI;
when '11 00'  // FMOV
  if fltsize != 16 && fltsize != intsize then UNDEFINED;
  op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
  part = 0;
when '11 01'  // FMOV D[1]
  if intsize != 64 || fltsize != 128 then UNDEFINED;
  op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
  part = 1;
  fltsize = 64;  // size of D[1] is 64
when '11 11'  // FJCVTZS
  if !HaveFJCVTZSExt() then UNDEFINED;
  rounding = FPRounding_ZERO;
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_FtoI_JS;
otherwise
  UNDEFINED;
end case;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTL, FCVTL2

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the FCVTL2 variant operates on the elements in the top 64 bits of the source register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

FCVTL2 <Vd>.<Ta>, <Vn>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR);

V[d] = result;
FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar half precision

FCVTMS <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar single-precision and double-precision

FCVTMS <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector half precision

FCVTMS $<Vd>.<T>,$ $<Vn>.<T>$

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

$<Hd>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Hn>$ Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

$<V>$ Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>$&lt;V&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

$<d>$ Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

$<n>$ Is the number of the SIMD&FP source register, encoded in the "Rn" field.

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.
```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);
V[d] = result;
```
FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 1 0</th>
<th>type</th>
<th>1 1 0</th>
<th>0 0 0</th>
<th>0 0 0</th>
<th>0 0 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmode</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{verbatim}
Half-precision to 32-bit (sf == 0 && type == 11) (ARMv8.2)

FCVTMS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11) (ARMv8.2)

FCVTMS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && type == 00)

FCVTMS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 00)

FCVTMS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 01)

FCVTMS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 01)

FCVTMS <Xd>, <Dn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
      rounding = FPDecodeRounding(rmode);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
      rounding = FPRoundingMode(FPCR);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
      rounding = FPRounding_TIEAWAY;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
      if fltsize != 16 && fltsize != intsize then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
      part = 0;
    when '11 01'        // FMOV D[1]
      if intsize != 64 || fltsize != 128 then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
      part = 1;
      fltsize = 64; // size of D[1] is 64
    when '11 11'        // FJCVTZS
      if !HaveFJCVTZSExt() then UNDEFINED;
      rounding = FPRounding_ZERO;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UNDEFINED;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTMU (vector)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>Scalar half precision (ARMv8.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Binary Representation]</td>
</tr>
<tr>
<td>Rn</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>o1</td>
</tr>
</tbody>
</table>

Scalar half precision

FCVTMU <Hd>, <Hn>

```assembly
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

<table>
<thead>
<tr>
<th>Scalar single-precision and double-precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Binary Representation]</td>
</tr>
<tr>
<td>Rn</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Rn</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>o1</td>
</tr>
</tbody>
</table>

Scalar single-precision and double-precision

FCVTMU <V>d>, <V>n>

```assembly
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

<table>
<thead>
<tr>
<th>Vector half precision (ARMv8.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Binary Representation]</td>
</tr>
<tr>
<td>Rn</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>o1</td>
</tr>
</tbody>
</table>
Vector half precision

FCVTMU \<Vd>\.<T>, \<Vn>\.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

\<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

\<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

\<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Half-precision to 32-bit (sf == 0 && type == 11) (ARMv8.2)

FCVTMU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11) (ARMv8.2)

FCVTMU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && type == 00)

FCVTMU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 00)

FCVTMU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 01)

FCVTMU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 01)

FCVTMU <Xd>, <Dn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
      rounding = FPDecodeRounding(rmode);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
      rounding = FPRoundingMode(FPCR);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
      rounding = FPRounding_TIEAWAY;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
      if fltsize != 16 && fltsize != intsize then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
      part = 0;
    when '11 01'        // FMOV D[1]
      if intsize != 64 || fltsize != 128 then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
      part = 1;
      fltsize = 64; // size of D[1] is 64;
    when '11 11'        // FJCVTZS
      if !HaveFJCVTZSExt() then UNDEFINED;
      rounding = FPRounding_ZERO;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UNDEFINED;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTN, FCVTN2

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

The FCVTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | Rn | Rd |

Vector single-precision and double-precision

FCVTN2 <Vd>.<Tb>, <Vn>.<Ta>

\[
\text{integer } d = \text{UInt}(Rd); \\
\text{integer } n = \text{UInt}(Rn); \\
\text{integer } esize = 16 << \text{UInt}(sz); \\
\text{integer } datasize = 64; \\
\text{integer } part = \text{UInt}(Q); \\
\text{integer } elements = \text{datasize} \div \text{esize};
\]

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>

Is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>

Is an arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR);
Vpart[d, part] = result;
FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

**Scalar half precision**

(ARMv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 0 0 1 1 1 1 0 0 1 1 0 1 0 1 0 1 0 1 0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>( U ) ( o2 ) ( o1 ) ( Rn ) ( Rd )</td>
</tr>
</tbody>
</table>

**Scalar single-precision and double-precision**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 0 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>( U ) ( o2 ) ( o1 ) ( Rn ) ( Rd )</td>
</tr>
</tbody>
</table>

**Vector half precision**

(ARMv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 1 1 1 0 0 0 1 1 1 0 0 1 1 0 1 0 1 0 1 0 1 0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>( U ) ( o2 ) ( o1 ) ( Rn ) ( Rd )</td>
</tr>
</tbody>
</table>
Vector half precision

FCVTNS <Vd>..<T>, <Vn>..<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPtoFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
   sf 0 0 1 1 1 0  type 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Rn  Rd
           rmode  opcode
```
Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

FCVTNS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

FCVTNS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && type == 00)

FCVTNS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 00)

FCVTNS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 01)

FCVTNS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 01)

FCVTNS <Xd>, <Dn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
when '00'  
  fltsize = 32;
when '01'  
  fltsize = 64;
when '10'  
  if opcode<2:1>:rmode != '11 01' then UNDEFINED;
  fltsize = 128;
when '11'  
  if HaveFPl6Ext() then 
    fltsize = 16;
  else
    UNDEFINED;
end;

case opcode<2:1>:rmode of
when '00 xx'        // FCVT[NPMZ][US]
  rounding = FPDecodeRounding(rmode);
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_FtoI;
when '01 00'        // [US]CVTF
  rounding = FPRoundingMode(FPCR);
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_ItoF;
when '10 00'        // FCVTA[US]
  rounding = FPRounding_TIEAWAY;
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_FtoI;
when '11 00'        // FMOV
  if fltsize != 16 || fltsize != intsize then UNDEFINED;
  op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
  part = 0;
when '11 01'        // FMOV D[1]
  if intsize != 64 || fltsize != 128 then UNDEFINED;
  op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
  part = 1;
  fltsize = 64; // size of D[1] is 64
when '11 11'        // FJCVTZS
  if !HaveFJCVTZSExt() then UNDEFINED;
  rounding = FPRounding_ZERO;
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_FtoI_JS;
otherwise
  UNDEFINED;
end;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

```
0 1 1 1 1 1 0 0 1 1 0 1 0 1 0 | Rn | Rd
U o2 o1
```

Scalar half precision

FCVTNU <Hd>, <Hn>

```c
if !HaveFP16Ext () then UNDEFINED;

integer d = UInt (Rd);
integer n = UInt (Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding (o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

(ARMv8.2)

```
0 1 1 1 1 1 0 0 0 0 0 1 1 0 1 0 1 0 | Rn | Rd
U o2 o1
```

Scalar single-precision and double-precision

FCVTNU <V><d>, <V><n>

```c
integer d = UInt (Rd);
integer n = UInt (Rn);

integer esize = 32 << UInt (sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding (o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

```
0 | Q 1 0 1 1 1 0 0 1 1 0 1 0 1 0 | Rn | Rd
U o2 o1
```

FCVTNU (vector)
Vector half precision

```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<V>`: Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>`: Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the 
SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-
purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FFPCR, the exception results in either a flag being 
set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an 
attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>type</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmode</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Half-precision to 32-bit (sf == 0 && type == 11) (ARMv8.2)

    FCVTNU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11) (ARMv8.2)

    FCVTNU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && type == 00)

    FCVTNU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 00)

    FCVTNU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 01)

    FCVTNU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 01)

    FCVTNU <Xd>, <Dn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

FPConvOp op;
FPRounding rounding;

boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
end;

case opcode<2:1>:rmode of
  when '00 xx'        // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00'        // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00'        // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00'        // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_FtoI else FPConvOp_MOV_ItoF;
    part = 0;
  when '11 01'        // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
  when '11 11'        // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
end;
otherwise
  UNDEFINED;
end;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = X[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = X[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTPS (vector)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

**(ARMv8.2)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>

U o2 o1

Scalar half precision

FCVTPS <Hd>, <Hn>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U o2 o1

Scalar single-precision and double-precision

FCVTPS <V>d>, <V>n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision

**(ARMv8.2)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U o2 o1
**Vector half precision**

FCVTPS `<Vd>.<T>, <Vn>.<T>`

if `!HaveFP16Ext()` then UNDEFINED;

integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;

integer esize = 16;
integer datasize = if `Q == '1'` then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = `FPDecodeRounding(o1:o2)`;
boolean unsigned = (U == '1');

**Vector single-precision and double-precision**

FCVTPS `<Vd>.<T>, <Vn>.<T>`

integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;

if `sz:Q == '10'` then UNDEFINED;
integer esize = 32 << `UInt(sz)`;
integer datasize = if `Q == '1'` then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = `FPDecodeRounding(o1:o2)`;
boolean unsigned = (U == '1');

**Assembler Symbols**

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPtoFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTPS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTPS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTPS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTPS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTPS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTPS <Xd>, <Dn>
```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  otherwise
    UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx'        // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00'        // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00'        // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00'        // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01'        // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11'        // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UNDEFINED;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTPU (vector)**

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

### Scalar half precision (ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 0 1 1 1 1 1 1 0 0 1 1 0 0 0 0 1 0 | Rn | Rd | U | o2 | o1 |

**FCVTPU** `<Hd>, <Hn>`

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 0 1 1 1 1 1 0 1 0 0 0 0 1 1 0 1 | Rn | Rd | U | o2 | o1 |

**FCVTPU** `<V><d>, <V><n>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector half precision (ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 0 Q 1 0 | 1 1 1 0 0 1 0 | Rn | Rd | U | o2 | o1 |

**FCVTPU** (vector)
Vector half precision

```
FCVTPU <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<V>`: Is a width specifier, encoded in "sz":
  
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>`: Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the vector half precision variant: is an arrangement specifier, encoded in “Q”:
  
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

  For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bias(datasize) operand = V[n];
bias(datasize) result;
bias(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Half-precision to 32-bit (sf == 0 && type == 11) (ARMv8.2)

FCVTPU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11) (ARMv8.2)

FCVTPU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && type == 00)

FCVTPU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 00)

FCVTPU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 01)

FCVTPU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 01)

FCVTPU <Xd>, <Dn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
      rounding = FPDecodeRounding(rmode);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
      rounding = FPRoundingMode(FPCR);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
      rounding = FPRounding_TIEAWAY;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
      if fltsize != 16 && fltsize != intsize then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp.Move_ItoF else FPConvOp.Move_FtoI;
      part = 0;
    when '11 01'        // FMOV D[1]
      if intsize != 64 || fltsize != 128 then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp.Move_ItoF else FPConvOp.Move_FtoI;
      part = 1;
      fltsize = 64;  // size of D[1] is 64
    when '11 11'        // FJCVTZS
      if !HaveFJCVTZSExt() then UNDEFINED;
      rounding = FPRounding_ZERO;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UNDEFINED;

Assembler Symbols

<Wd>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>  Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTXN, FCVTXN2

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD&FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The FCVTXN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTXN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | Rn | Rd |

Scalar

FCVTXN <Vb><d>, <Va><n>

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0;

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | Rn | Rd |

Vector

FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = UInt(Q);

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”: 
<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “sz;Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vb>  Is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
</tr>
</tbody>
</table>

<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va>  Is the source width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<n>  Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();

bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPCConvert(Elem[operand, e, 2*esize], FPCR, FPRounding_ODD);

Vpart[d, part] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00be8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!= 0000</td>
<td>immb</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>immh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar

FCVTZS <V<d>, <V<n>, #<fbits>

integer d = _UInt(Rd);
integer n = _UInt(Rn);
if immh == '000x' || (immh == '001x' && !_HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;
integer fracbits = (esize * 2) - _UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

Vector

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!= 0000</td>
<td>immb</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>immh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector

FCVTZS <Vd>, <T>, <Vn>, <T>, #<fbits>

integer d = _UInt(Rd);
integer n = _UInt(Rn);
if immh == '000x' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !_HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer fracbits = (esize * 2) - _UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

Assembler Symbols

<V> Is a width specifier, encoded in “immh”: 
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>0000</th>
<th>0001</th>
<th>001x</th>
<th>01xx</th>
<th>1xxx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>4H</td>
<td>8H</td>
<td>2S</td>
<td>4S</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>RESERVED</td>
<td></td>
<td></td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits>
For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>000x</th>
<th>001x</th>
<th>01xx</th>
<th>1xxx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESERVED</td>
<td>(32-Uint(immh:immb))</td>
<td>(64-Uint(immh:immb))</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>0000</th>
<th>0001</th>
<th>001x</th>
<th>01xx</th>
<th>1xxx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SEE Advanced SIMD modified immediate</td>
<td>RESERVED</td>
<td>(32-Uint(immh:immb))</td>
<td>(64-Uint(immh:immb))</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bites(datasize) result;
bites(esize) element;
for e = 0 to elements-1
   element = Elem[operand, e, esize];
   elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);
V[d] = result;
FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

```
0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 0
U  o2  o1
```

Scalar half precision

```
FCVTZS <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

```
0 1 1 1 1 0 1 0 0 0 0 1 1 0 1 1 0
U  o2  o1
```

Scalar single-precision and double-precision

```
FCVTZS <V>d>, <V>n
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

```
0 Q 0 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 0
U  o2  o1
```

Vector half precision

```
```
Vector half precision

FCVTZS <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 1 0</th>
<th>type</th>
<th>0 1 1 0 0 0</th>
<th>scale</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rmode</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Half-precision to 32-bit (sf == 0 && type == 11) (ARMv8.2)

    FCVTZS <Wd>, <Hn>, #<fbits>

Half-precision to 64-bit (sf == 1 && type == 11) (ARMv8.2)

    FCVTZS <Xd>, <Hn>, #<fbits>

Single-precision to 32-bit (sf == 0 && type == 00)

    FCVTZS <Wd>, <Sn>, #<fbits>

Single-precision to 64-bit (sf == 1 && type == 00)

    FCVTZS <Xd>, <Sn>, #<fbits>

Double-precision to 32-bit (sf == 0 && type == 01)

    FCVTZS <Wd>, <Dn>, #<fbits>

Double-precision to 64-bit (sf == 1 && type == 01)

    FCVTZS <Xd>, <Dn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer f甘肃省 = 64 - UInt(scale);
FPRoundingOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            f甘肃省size = 16;
        else
            UNDEFINED;
    end;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
    when '00 11'        // FCVTZ
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPCvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPCvOp_CVT_ItoF;
    otherwise
        UNDEFINED;
end;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".

For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
    V[d] = fltval;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rec3; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| sf | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rn | Rd |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```
Half-precision to 32-bit (sf == 0 && type == 11) (ARMv8.2)

FCVTZS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11) (ARMv8.2)

FCVTZS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && type == 00)

FCVTZS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 00)

FCVTZS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 01)

FCVTZS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 01)

FCVTZS <Xd>, <Dn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;

boolean unsigned;
integer part;

case type of
when '00'
  fltsize = 32;
when '01'
  fltsize = 64;
when '10'
  if opcode<2:1>:rmode != '11 01' then UNDEFINED;
  fltsize = 128;
when '11'
  if HaveFP16Ext() then
    fltsize = 16;
  else
    UNDEFINED;
end;

case opcode<2:1>:rmode of
when '00 xx'        // FCVT[NPMZ][US]
  rounding = FPDecodeRounding(rmode);
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_FtoI;
when '01 00'        // [US]CVTF
  rounding = FPRoundingMode(FPCR);
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_ItoF;
when '10 00'        // FCVTA[US]
  rounding = FPRounding_TIEAWAY;
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_FtoI;
when '11 00'        // FMOV
  if fltsize != 16 && fltsize != intsize then UNDEFINED;
  op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
  part = 0;
when '11 01'        // FMOV D[1]
  if intsize != 64 || fltsize != 128 then UNDEFINED;
  op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
  part = 1;
  fltsize = 64; // size of D[1] is 64
when '11 11'        // FJCVTZS
  if !HaveFJCVTZSExt() then UNDEFINED;
  rounding = FPRounding_ZERO;
  unsigned = (opcode<0> == '1');
  op = FPConvOp_CVT_FtoI_JS;
otherwise
  UNDEFINED;
end;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = \texttt{V}[n];
        intval = \texttt{FPToFixed}(fltval, 0, unsigned, FPCR, rounding);
        \texttt{X}[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = \texttt{X}[n];
        fltval = \texttt{FixedToFP}(intval, 0, unsigned, FPCR, rounding);
        \texttt{V}[d] = fltval;
    when FPConvOp_MOV_FtoI
        fltval = \texttt{Vpart}[n,part];
        intval = \texttt{ZeroExtend}(fltval, intsize);
        \texttt{X}[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = \texttt{X}[n];
        fltval = intval<fltsize-1:0>;
        \texttt{Vpart}[d,part] = fltval;
    when FPConvOp_CVT_FtoI_JS
        fltval = \texttt{V}[n];
        intval = \texttt{FPToFixedJS}(fltval, FPCR, TRUE);
        \texttt{X}[d] = \texttt{ZeroExtend}(intval<31:0>, 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTZU (vector, fixed-point)**

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | !=0000 | immh | 1  | 1  | 1  | 1  | 1  | 1  | Rd  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Vector**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | !=0000 | immh | 1  | 1  | 1  | 1  | 1  | 1  | Rd  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Asmmler Symbols**

<\V> Is a width specifier, encoded in “immh”:

```plaintext
Integer d = UInt(Rd);
Integer n = UInt(Rn);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
Integer esize = if immh == 'lxxx' then 64 else if immh == '01xx' then 32 else 16;
Integer datasize = esize;
Integer elements = 1;
Integer frachbits = (esize * 2) - UInt(immh:immb);
Boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```
<V>  
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>  
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  
Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>  
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits>  
For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
b(x)ts(datasize) result;
b(x)ts(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPT0Fixed(element, fracbits, unsigned, FPCR, rounding);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision (ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  |  |  |  |  |
| U  | o2 | o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Scalar half precision

FCVTZU <Hd>, <Hn>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  |
| U  | o2 | o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Scalar single-precision and double-precision

FCVTZU <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision (ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  |
| U  | o2 | o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Vector half precision

FCVTU <Vd>.,<T>, <Vn>..<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPtoFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>type</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>scale</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmode</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Half-precision to 32-bit (sf == 0 && type == 11) (ARMv8.2)

FCVTZU <Wd>, <Hn>, #<fbits>

Half-precision to 64-bit (sf == 1 && type == 11) (ARMv8.2)

FCVTZU <Xd>, <Hn>, #<fbits>

Single-precision to 32-bit (sf == 0 && type == 00)

FCVTZU <Wd>, <Sn>, #<fbits>

Single-precision to 64-bit (sf == 1 && type == 00)

FCVTZU <Xd>, <Sn>, #<fbits>

Double-precision to 32-bit (sf == 0 && type == 01)

FCVTZU <Wd>, <Dn>, #<fbits>

Double-precision to 64-bit (sf == 1 && type == 01)

FCVTZU <Xd>, <Dn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
when '00' fltsize = 32;
when '01' fltsize = 64;
when '10' UNDEFINED;
when '11'
    if HaveFP16Ext() then
        fltsize = 16;
    else
        UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
when '00 11' // FCVTZ
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
otherwise
    UNDEFINED;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".

For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
        V[d] = fltval;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Half-precision to 32-bit (sf == 0 && type == 11) (ARMv8.2)

FCVTZU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11) (ARMv8.2)

FCVTZU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && type == 00)

FCVTZU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 00)

FCVTZU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 01)

FCVTZU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 01)

FCVTZU <Xd>, <Dn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  otherwise
    UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx'        // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00'        // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00'        // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00'        // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01'        // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64;  // size of D[1] is 64
  when '11 11'        // FJCVTZS
    if HaveFJCVTZSExt() then UNDEFINED;
    rounding = FPRounding.ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UNDEFINED;

Assembler Symbols

<Wd>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>  Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        fltval = Vpart[n,part];
        intval = ZeroExtend(fltval, intsize);
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        fltval = intval<fltsize-1:0>;
        Vpart[d,part] = fltval;
    when FPConvOp_CVT_FtoI_JS
        fltval = V[n];
        intval = FPToFixedJS(fltval, FPCR, TRUE);
        X[d] = ZeroExtend(intval<31:0>, 64);

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FDIV (vector)

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | 1  | 1  | 1  | Rn | 0  | Rd |

Half-precision

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 1  | 1  | 1  | Rd |

Single-precision and double-precision

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDIV(element1, element2, FPCR);
V[d] = result;
```
FDIV (scalar)

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 1 0 type 1 | Rd 0 0 0 1 1 0 | Rn | Rd

Half-precision (type == 11) (ARMv8.2)

FDIV <Hd>, <Hn>, <Hm>

Single-precision (type == 00)

FDIV <Sd>, <Sn>, <Sm>

Double-precision (type == 01)

FDIV <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand1 = \texttt{V}[n];
bits(datasize) operand2 = \texttt{V}[m];

result = \texttt{FPDiv}(operand1, operand2, FPCR);
\texttt{V}[d] = result;
FJCVTZS

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo $2^{32}$, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in $FPSCR$, the exception results in either a flag being set in $FPSCR$ or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Double-precision to 32-bit

(ARMv8.3)

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf |  0 |  0 |  0 |  1 |  1 |  1 |  1 |  0 |  0 |  1 |  1 |  1 |  1 |  1 |  1 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |
| sf |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Double-precision to 32-bit

FJCVTZS <Wd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;
case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
      rounding = FPDecodeRounding(rmode);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
      rounding = FPRoundingMode(FPCR);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
      rounding = FPRounding_TIEAWAY;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
      if fltsize != 16 && fltsize != intsize then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
      part = 0;
    when '11 01'        // FMOV D[1]
      if intsize != 64 || fltsize != 128 then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
      part = 1;
      fltsize = 64; // size of D[1] is 64
    when '11 11'        // FJCVTZS
      if !HaveFJCVTZSExt() then UNDEFINED;
      rounding = FPRounding_ZERO;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UNDEFINED;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        fltval = Vpart[n,part];
        intval = ZeroExtend(fltval, intsize);
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        fltval = intval<fltsize-1:0>;
        Vpart[d,part] = fltval;
    when FPConvOp_CVT_FtoI_JS
        fltval = V[n];
        intval = FPToFixedJS(fltval, FPCR, TRUE);
        X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPSR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | Rm | 0  | Ra | 0  | Rn | 0  | Rd |

**Half-precision (type == 11)**

(ARMv8.2)

FMADD <Hd>, <Hn>, <Hm>, <Ha>

**Single-precision (type == 00)**

FMADD <Sd>, <Sn>, <Sm>, <Sa>

**Double-precision (type == 01)**

FMADD <Dd>, <Dn>, <Dm>, <Da>

```plaintext
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
switch (type) {
    case '00': datasize = 32;
    case '01': datasize = 64;
    case '10': UNDEFINED;
    case '11':
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
        break;
}

boolean opa_neg = (o1 == '1');
boolean opl_neg = (o0 != o1);
```

**Assembler Symbols**

- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>`: Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Dm>`: Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Da>`: Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Hm>`: Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Ha>`: Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- `<Sd>`: Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>`: Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if opa_neg then operanda = FPNeg(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 1 | 0 | 1 | Rd | |

**Half-precision**

\[ \text{FMAX} \langle \text{Vd} \rangle, \langle \text{T} \rangle, \langle \text{Vn} \rangle, \langle \text{T} \rangle, \langle \text{Vm} \rangle, \langle \text{T} \rangle \]

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

**Single-precision and double-precision**

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 1 | 0 | 1 | Rd | |

**Single-precision and double-precision**

\[ \text{FMAX} \langle \text{Vd} \rangle, \langle \text{T} \rangle, \langle \text{Vn} \rangle, \langle \text{T} \rangle, \langle \text{Vm} \rangle, \langle \text{T} \rangle \]

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

**Assembler Symbols**

\(<\text{Vd}> \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}\)

\(<\text{T}> \quad \text{For the half-precision variant: is an arrangement specifier, encoded in "Q"; for the single-precision and double-precision variant: is a size specifier, encoded in "sz".}\)
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR);
V[d] = result;
```
FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 1  | 0  | type | 1  | Rm | 0  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |

op

Half-precision (type == 11)
(ARMv8.2)

FMAX <Hd>, <Hn>, <Hm>

Single-precision (type == 00)

FMAX <Sd>, <Sn>, <Sm>

Double-precision (type == 01)

FMAX <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;

case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11' if HaveFP16Ext() then
datasize = 16;
  else
    UNDEFINED;

FPMaxMinOp operation;

case op of
  when '00' operation = FPMaxMinOp_MAX;
  when '01' operation = FPMaxMinOp_MIN;
  when '10' operation = FPMaxMinOp_MAXNUM;
  when '11' operation = FPMaxMinOp_MINNUM;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
    when FPMaxMinOp_MAX  result = FPMax(operand1, operand2, FPCR);
    when FPMaxMinOp_MIN  result = FPMin(operand1, operand2, FPCR);
    when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
    when FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNM (vector)

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision
(ARMv8.2)

\[
\begin{array}{cccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & Rm & 0 & 0 & 0 & 0 & 1 & Rn & Rd \\
\hline
U & a & \\
\end{array}
\]

### Single-precision and double-precision

\[
\begin{array}{cccccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 0 & 0 & \text{sz} & 1 & Rm & 1 & 1 & 0 & 0 & 0 & 1 & Rn & Rd \\
\hline
\end{array}
\]

### Single-precision and double-precision

\[
\begin{array}{cccccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 0 & 0 & \text{sz} & 1 & Rm & 1 & 1 & 0 & 0 & 0 & 1 & Rn & Rd \\
\hline
U & 01 & \\
\end{array}
\]

### Assembler Symbols

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q";

\(<Vn>\), \(<Vm>\) For the single-precision and double-precision variant.
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);
V[d] = result;
```
FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision (type == 11)
(ARMv8.2)

FMAXNM <Hd>, <Hn>, <Hm>

Single-precision (type == 00)

FMAXNM <Sd>, <Sn>, <Sm>

Double-precision (type == 01)

FMAXNM <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
data size = 16;
        else
            UNDEFINED;
FPMaxMinOp operation;
case op of
    when '00' operation = FPMaxMinOp_MAX;
    when '01' operation = FPMaxMinOp_MIN;
    when '10' operation = FPMaxMinOp_MAXNUM;
    when '11' operation = FPMaxMinOp_MINNUM;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
  when FPMaxMinOp_MAX      result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN      result = FPMin(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM   result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM   result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td>o1</td>
<td>sz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision

FMAXNMP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

**Single-precision and double-precision**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Single-precision and double-precision

FMAXNMP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

**Assembler Symbols**

For the half-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U

a

**FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

### Single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in “Q”:

\[
\begin{array}{c|c}
\text{Q} & \langle T \rangle \\
\hline
0 & 4H \\
1 & 8H \\
\end{array}
\]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

\[
\begin{array}{c|c|c}
sz & Q & \langle T \rangle \\
\hline
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & \text{RESERVED} \\
1 & 1 & 2D \\
\end{array}
\]

<\text{T}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{Vn}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = \text{V}[n];
bits(datasize) operand2 = \text{V}[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = \text{Elem}[concat, 2*e, esize];
        element2 = \text{Elem}[concat, (2*e)+1, esize];
    else
        element1 = \text{Elem}[operand1, e, esize];
        element2 = \text{Elem}[operand2, e, esize];
    if minimum then
        \text{Elem}[result, e, esize] = \text{FPMinNum}(element1, element2, FPCR);
    else
        \text{Elem}[result, e, esize] = \text{FPMaxNum}(element1, element2, FPCR);
\text{V}[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to FMAX(scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | R  | n  | Rd |

\( o1 \)

Half-precision

FMAXNMV <V><d>, <Vn>..<T>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | R  | n  | Rd |

\( o1 \)

Single-precision and double-precision

FMAXNMV <V><d>, <Vn>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED; // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

Assembler Symbols

\(<V>\)

For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

...
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | | Rn | Rd |
| o1 | sz |

#### FMAXP <\V><d>, <\Vn>.<T>

if ![HaveFP16Ext](#) then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

### Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | | Rn | Rd |

#### FMAXP <\V><d>, <\Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

### Assembler Symbols

\(<\V>\) For the half-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>(&lt;\V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

```
CheckFPAdvSIMDEnabled64();
bite(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|------------------|
| 0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 1 | 0 | 1 | Rn | Rd |
| U | o1 |

Half-precision

FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|------------------|
| 0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 1 | 0 | 1 | Rn | Rd |
| U | o1 |

Single-precision and double-precision

FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>  For the half-precision variant: is an arrangement specifier, encoded in “Q”:
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bvbits(datasize) operand1 = V[n];
bvbits(datasize) operand2 = V[m];
bvbits(datasize) result;
bvbits(2*datasize) concat = operand2:operand1;
bvbits(esize) element1;
bvbits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 01 |

| Rn | Rd |

#### Half-precision

FMAXV `<V><d>, <Vn>.<T>`

if !`HaveFP16Ext` () then UNDEFINED;

integer d = `UInt`(Rd);
integer n = `UInt`(Rn);

integer esize = 16;
integer datasize = if `Q` == '1' then 128 else 64;
integer elements = datasize DIV esize;

`ReduceOp` op = if `o1` == '1' then `ReduceOp_FMIN` else `ReduceOp_FMAX`;

### Single-precision and double-precision

| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 01 |

| Rn | Rd |

#### Single-precision and double-precision

FMAXV `<V><d>, <Vn>.<T>`

integer d = `UInt`(Rd);
integer n = `UInt`(Rn);

if `sz:Q` != '01' then UNDEFINED;

integer esize = 32 << `UInt`(sz);
integer datasize = if `Q` == '1' then 128 else 64;
integer elements = datasize DIV esize;

`ReduceOp` op = if `o1` == '1' then `ReduceOp_FMIN` else `ReduceOp_FMAX`;

### Assembler Symbols

<table>
<thead>
<tr>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
</table>

For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th><code>sz</code></th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMIN (vector)

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | 0  | 0  | 1  | 1  | 0  | 1  | Rn | Rd |

Half-precision

\[ \text{FMIN} \langle Vd \rangle \langle T \rangle, \langle Vn \rangle \langle T \rangle, \langle Vm \rangle \langle T \rangle \]

\[ \text{if } !\text{HaveFP16Ext()} \text{ then UNDEFINED;} \]

\[ \text{integer } d = \text{UInt}(\text{Rd}); \]
\[ \text{integer } n = \text{UInt}(\text{Rn}); \]
\[ \text{integer } m = \text{UInt}(\text{Rm}); \]
\[ \text{integer esize} = 16; \]
\[ \text{integer datasize} = \text{if } Q = '1' \text{ then } 128 \text{ else } 64; \]
\[ \text{integer elements} = \text{datasize} \div \text{esize}; \]

\[ \text{boolean pair} = (U \text{ } == \text{ } '1'); \]
\[ \text{boolean minimum} = (o1 \text{ } == \text{ } '1'); \]

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 1  | Rn | Rd |

Single-precision and double-precision

\[ \text{FMIN} \langle Vd \rangle \langle T \rangle, \langle Vn \rangle \langle T \rangle, \langle Vm \rangle \langle T \rangle \]

\[ \text{integer } d = \text{UInt}(\text{Rd}); \]
\[ \text{integer } n = \text{UInt}(\text{Rn}); \]
\[ \text{integer } m = \text{UInt}(\text{Rm}); \]
\[ \text{if } sz\text{ }Q \text{ } == \text{ } '10' \text{ } \text{then UNDEFINED}; \]
\[ \text{integer esize} = 32 \text{ } \text{shift} \text{ } \text{UInt}(sz); \]
\[ \text{integer datasize} = \text{if } Q = '1' \text{ then } 128 \text{ else } 64; \]
\[ \text{integer elements} = \text{datasize} \div \text{esize}; \]

\[ \text{boolean pair} = (U \text{ } == \text{ } '1'); \]
\[ \text{boolean minimum} = (o1 \text{ } == \text{ } '1'); \]

Assembler Symbols

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in “Q”: 
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMIN (scalar)

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision (type == 11) (ARMv8.2)

\[
\text{FMIN}<\text{Hd}>,<\text{Hn}>,<\text{Hm}>
\]

Single-precision (type == 00)

\[
\text{FMIN}<\text{Sd}>,<\text{Sn}>,<\text{Sm}>
\]

Double-precision (type == 01)

\[
\text{FMIN}<\text{Dd}>,<\text{Dn}>,<\text{Dm}>
\]

integer \( \text{d} = \text{Uint}(\text{Rd}); \)
integer \( \text{n} = \text{Uint}(\text{Rn}); \)
integer \( \text{m} = \text{Uint}(\text{Rm}); \)

integer \( \text{datasize}; \)
case type of
when '00' \( \text{datasize} = 32; \)
when '01' \( \text{datasize} = 64; \)
when '10' \( \text{UNDEFINED}; \)
when '11'
if \( \text{HaveFP16Ext}() \) then
\( \text{datasize} = 16; \)
else
\( \text{UNDEFINED}; \)
endcase

\( \text{FPMaxMinOp} \) operation;
case op of
when '00' \( \text{operation} = \text{FPMaxMinOp\_MAX}; \)
when '01' \( \text{operation} = \text{FPMaxMinOp\_MIN}; \)
when '10' \( \text{operation} = \text{FPMaxMinOp\_MAXNUM}; \)
when '11' \( \text{operation} = \text{FPMaxMinOp\_MINNUM}; \)
endcase

Assembler Symbols

\(<\text{Dd}>\)  Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<\text{Dn}>\)  Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
\(<\text{Dm}>\)  Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
\(<\text{Hd}>\)  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<\text{Hn}>\)  Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
\(<\text{Hm}>\)  Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
\(<\text{Sd}>\)  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<\text{Sn}>\)  Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
  when FPMaxMinOp_MAX result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN result = FPMin(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMNNM (vector)

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------|-----------------------------|-----------------------------|
| 0  Q 0 1 1 1 0 1 1 0  Rm 0 0 0 0 0 0 1  Rn  Rd |
| 0  a |
```

**Half-precision**

FMNNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```bash
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------|-----------------------------|-----------------------------|
| 0  Q 0 1 1 1 0 1 1 0  sz 1  Rm 1 1 0 0 0 0 1  Rn  Rd |
| 0 01 |
```

**Single-precision and double-precision**

FMNNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```bash
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Assembler Symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in “Q”: 
  ```bash
  if !HaveFP16Ext() then UNDEFINED;
  ```
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINNM (scalar)

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to \texttt{FMIN (scalar)}.

This instruction can generate a floating-point exception. Depending on the settings in \texttt{FPCR}, the exception results in either a flag being set in \texttt{FPSR} or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

\textbf{Half-precision (type == 11)}

(ARMv8.2)

\begin{verbatim}
FMINNM <Hd>, <Hn>, <Hm>
\end{verbatim}

\textbf{Single-precision (type == 00)}

\begin{verbatim}
FMINNM <Sd>, <Sn>, <Sm>
\end{verbatim}

\textbf{Double-precision (type == 01)}

\begin{verbatim}
FMINNM <Dd>, <Dn>, <Dm>
\end{verbatim}

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);

integer datasize;
\textbf{case} type \textbf{of}
 \textbf{case} op \textbf{of}
when '00' datasize = 32;
when '01' datasize = 64;
when '10' UNDEFINED;
when '11' \textbf{if} \texttt{HaveFP16Ext()} \textbf{then}
\hspace{1em}datasize = 16;
\hspace{1em}\textbf{else}
\hspace{2em}UNDEFINED;
\texttt{FPMaxMinOp} operation;
\texttt{FMaxMinOp} MAX;
when '01' operation = \texttt{FMaxMinOp\_MIN};
when '10' operation = \texttt{FMaxMinOp\_MAXNUM};
when '11' operation = \texttt{FMaxMinOp\_MINNUM};

\textbf{Assembler Symbols}

\begin{verbatim}
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\end{verbatim}
Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
    when FPMaxMinOp_MAX result = FMax(operand1, operand2, FPCR);
    when FPMaxMinOp_MIN result = FMin(operand1, operand2, FPCR);
    when FPMaxMinOp_MAXNUM result = FMaxNum(operand1, operand2, FPCR);
    when FPMaxMinOp_MINNUM result = FMinNum(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00ber8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

<table>
<thead>
<tr>
<th>Encoding of FMINNMP (scalar) instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Half-precision</strong></td>
</tr>
<tr>
<td>(ARMv8.2)</td>
</tr>
<tr>
<td>01 11 11 1 0 0 0 0 0 1 1 0 0 1 0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>o1</td>
</tr>
</tbody>
</table>

**Half-precision**

FMINNMP <V><d>, <Vn>.<T>

```plaintext
if !HaveFP16Ext () then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = esize * 2;
integer elements = 2;
ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

<table>
<thead>
<tr>
<th>Encoding of FMINNMP (scalar) instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single-precision and double-precision</strong></td>
</tr>
<tr>
<td>01 11 11 1 0 1 1 0 0 0 0 0 1 1 0 0 1 0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>o1</td>
</tr>
</tbody>
</table>

**Single-precision and double-precision**

FMINNMP <V><d>, <Vn>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;
ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

**Assembler Symbols**

For the half-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | |
| U  | a  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

**Single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | |
| sz | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U  | o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0     4H</td>
</tr>
<tr>
<td>1     8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz Q &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 2S</td>
</tr>
<tr>
<td>0 1 4S</td>
</tr>
<tr>
<td>1 0 RESERVED</td>
</tr>
<tr>
<td>1 1 2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
b bits(datasize) operand1 = V[n];
b bits(datasize) operand2 = V[m];
b bits(datasize) result;
b bits(2*datasize) concat = operand2:operand1;
b bits(esize) element1;
b bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);
    V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMNNMV

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to \textit{FMIN (scalar)}.

This instruction can generate a floating-point exception. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR} or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR_EL1}, \textit{CPTR_EL2}, and \textit{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \textit{Half-precision} and \textit{Single-precision and double-precision}.

\textbf{Half-precision (ARMv8.2)}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |
| o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

\textbf{Single-precision and double-precision}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 1  | 1  | 0  | 1  | sz | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |
| o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

\textbf{Assembler Symbols}

- <V> For the half-precision variant: is the destination width specifier, H.
- For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”: \textit{FMINNMV <V><d>, <Vn>.<T>}

```plaintext
if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

```plaintext
if sz:Q != '01' then UNDEFINED; // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |
| o1 | sz |

Half-precision

FMINP <V<d>, <Vn>.<T>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |
| o1 |

Single-precision and double-precision

FMINP <V<d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINP (vector)

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision (ARMv8.2)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 1  | Rd |
|   | U  | o1 |

Half-precision

FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Single-precision and double-precision

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 1  | Rd |
|   | U  | o1 |

Single-precision and double-precision

FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

**ARMv8.2**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | Rn  | Rd  |

#### Half-precision

FMINV `<V>d, <Vn>.

```plaintext
if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

### Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | Rn  | Rd  |

#### Single-precision and double-precision

FMINV `<V>d, <Vn>.

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

### Assembler Symbols

- `<V>`: For the half-precision variant: is the destination width specifier, H.
  - For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

---

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in `FPSCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

(ARMv8.2)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | L  | M  | Rm | 0  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

o2
```

Scalar, half-precision

```
FMLA <Hd>, <Hn>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Scalar, single-precision and double-precision

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | sz | L  | M  | Rm | 0  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

o2
```
Scalar, single-precision and double-precision

FMLA \(<\text{V}><d>, <\text{V}><n>, <\text{V}m><\text{T}s>[<\text{index}>]\)\]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');

Vector, half-precision
(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector, half-precision

FMLA \(<\text{V}d>.<\text{T}>, <\text{V}n>.<\text{T}>, <\text{V}m>.\text{H}[<\text{index}>]\)\]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Vector, single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>sz</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector, single-precision and double-precision

FMLA <Vd>,<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(usize) element1;
bits(usize) element2 = Elem[operand2, index, usize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, usize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, usize] = FPMulAdd(Elem[operand3, e, usize], element1, element2, FPCR);
V[d] = result;
```
FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 Q 0 0 1 1 0 | 0 1 0          | Rm             | 0 0 0 0 1 1    | Rd             |

Half-precision

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');

Single-precision and double-precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 Q 0 0 1 1 0 | 0 1 1 0        | Rm             | 1 1 0 0 1 1    | Rd             |

Single-precision and double-precision

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

| sz | Q | <T>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>2S</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>4S</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>2D</td>
<td></td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

It has encodings from 2 classes: FMLAL and FMLAL2

### FMLAL

**FMLAL (ARMv8.2)**

```
0 | Q | 0 | 0 | 1 | 1 | 1 | 1 | 0 | L | M | Rm | 0 | 0 | 0 | 0 | H | 0 | Rn | Rd
```

### FMLAL2

**FMLAL2 (ARMv8.2)**

```
0 | Q | 1 | 0 | 1 | 1 | 1 | 1 | 0 | L | M | Rm | 1 | 0 | 0 | 0 | H | 0 | Rn | Rd
```

### FMLAL

```
FMLAL <Vd>,<Ta>, <Vn>,<Tb>, <Vm>.H[<index>]
```

```java
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);     // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);
integer esize = 32;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

### FMLAL2

```
FMLAL2 <Vd>,<Ta>, <Vn>,<Tb>, <Vm>.H[<index>]
```

```java
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);     // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```
Assembler Symbols

\(<V_d>\) Is the name of the SIMD\&FP destination register, encoded in the "Rd" field.

\(<T_a>\) Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th></th>
<th>Q&lt;(T_a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

\(<V_{n}>\) Is the name of the first SIMD\&FP source register, encoded in the "Rn" field.

\(<T_b>\) Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th></th>
<th>Q&lt;(T_b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

\(<V_{m}>\) Is the name of the second SIMD\&FP source register, encoded in the "Rm" field.

\(<\text{index}>\) Is the element index, encoded in the "H:L:M" fields.

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize DIV 2) operand1 = Vpart[n,part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bots(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMLAL, FMLAL2 (vector)**

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

It has encodings from 2 classes: FMLAL and FMLAL2

### FMLAL

**FMLAL (ARMv8.2)**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
</tr>
</tbody>
</table>

| S | sz |

**FMLAL**

FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```plaintext
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;
```

### FMLAL2

**FMLAL2 (ARMv8.2)**

| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | Rm | 1 | 1 | 0 | 0 | 1 | 1 | Rn | Rd |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| S | sz |

**FMLAL2**

FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```plaintext
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;
```

### Assembler Symbols

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n,part];
bits(datasize DIV 2) operand2 = Vpart[m,part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result,e,esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLS (by element)

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | L  | M  | Rm | 0  | 1  | 0  | 1  | H  | 0  | Rn | Rd |

Scalar, single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | sz | L  | M  | Rm | 0  | 1  | 0  | 1  | H  | 0  | Rn | Rd |

FMLS <Hd>, <Hn>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
Scalar, single-precision and double-precision

FMLS <V><d>, <V><n>, <Vm><Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');

Vector, half-precision
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | L  | M  | Rm | 0  | 1  | 0  | 1  | H  | 0  | Rn | Rd |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

o2

Vector, half-precision

FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Vector, single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 1  | sz | L  | M  | Rm | 0  | 1  | 0  | 1  | H  | 0  | Rn | Rd |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

o2

FMLS (by element)
Vector, single-precision and double-precision

FMLS <Vd>,<T>, <Vn>,<T>, <Vm>,<Ts>[<index>]

integer idxdsizes = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:Q of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```
FMLS (vector)

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 1 | 1 | Rd |

\[
\text{FMLS} <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
\]

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');

### Single-precision and double-precision

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 1 | 1 | Rn | 1 | 1 | 0 | 0 | 1 | 1 | Rd |

\[
\text{FMLS} <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
\]

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');

### Assembler Symbols

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

\[
\text{<Vd>} \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[
\text{<T>} \quad \text{For the half-precision variant: is an arrangement specifier, encoded in “Q”:
}
\]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

\[
\text{<Vd>} \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[
\text{<T>} \quad \text{For the half-precision variant: is an arrangement specifier, encoded in “Q”:
}
\]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it. ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL (ARMv8.2)

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```plaintext
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLSL2 (ARMv8.2)

FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```plaintext
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```
Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<index> Is the element index, encoded in the "H:L:M" fields.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize DIV 2) operand1 = Vpart[n,part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
FMLSL, FMLSL2 (vector)

Floating-point fused Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL
(ARMv8.2)

| Field  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|--------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Rm     | Q | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | S  | 0  | 1 | 1 | 1 | 0 | 1 |   | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rn     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rd     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FMLSL

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext () then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;

FMLSL2
(ARMv8.2)

| Field  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|--------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Rm     | Q | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | S  | 0  | 1 | 1 | 0 | 0 | 1 |   | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rn     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rd     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FMLSL2

FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext () then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
<td></td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
<td></td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n,part];
bits(datasize DIV 2) operand2 = Vpart[m,part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result,e,esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
```

0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | a | b | c | 1 | 1 | 1 | 1 | 1 | d | e | f | g | h | Rd

Half-precision

FMOV <Vd>.<T>, #<imm>

```java
if !HaveFP16Ext() then UNDEFINED;

integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;

imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,2):imm8<5:0>:Zeros(6);

imm = Replicate(imm16, datasize DIV 16);
```

Single-precision and double-precision

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
```

0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | a | b | c | 1 | 1 | 1 | 1 | 0 | d | e | f | g | h | Rd
cmode
Single-precision (op == 0)

FMOV <Vd>,<T>, #<imm>

Double-precision (Q == 1 && op == 1)

FMOV <Vd>.2D, #<imm>

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in A64 floating-point instructions.

Operation

CheckFPAdvSIMDEnabled64();

V[rd] = imm;
FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

|31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|0 | 0 | 1 | 1 | 1 | 0 | type | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Rn | Rd | opc |

Half-precision (type == 11)
(ARMv8.2)

\[
\text{FMOV \(<Hd>, \<Hn>\)}
\]

Single-precision (type == 00)

\[
\text{FMOV \(<Sd>, \<Sn>\)}
\]

Double-precision (type == 01)

\[
\text{FMOV \(<Dd>, \<Dn>\)}
\]

```plaintext
type = 00
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
end;

FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
end;
```

Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
    when FPUnaryOp_MOV result = operand;
    when FPUnaryOp_ABS result = FPAbs(operand);
    when FPUnaryOp_NEG result = FPNeg(operand);
    when FPUnaryOp_SQRT result = FPSqrt(operand, FPCR);

V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| sf | 0 | 0 | 1 | 1 | 1 | 0 | type | 1 | 0 | x | 1 | 1 | x | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
```

- sf: Sign flag
- type: Move type
- Rn: Source register
- Rd: Destination register
Half-precision to 32-bit (sf == 0 && type == 11 && rmode == 00 && opcode == 110)
(ARMv8.2)

FMOV <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && type == 11 && rmode == 00 && opcode == 110)
(ARMv8.2)

FMOV <Xd>, <Hn>

32-bit to half-precision (sf == 0 && type == 11 && rmode == 00 && opcode == 111)
(ARMv8.2)

FMOV <Hd>, <Wn>

32-bit to single-precision (sf == 0 && type == 00 && rmode == 00 && opcode == 111)

FMOV <Sd>, <Wn>

Single-precision to 32-bit (sf == 0 && type == 00 && rmode == 00 && opcode == 110)

FMOV <Wd>, <Sn>

64-bit to half-precision (sf == 1 && type == 11 && rmode == 00 && opcode == 111)
(ARMv8.2)

FMOV <Hd>, <Xn>

64-bit to double-precision (sf == 1 && type == 01 && rmode == 00 && opcode == 111)

FMOV <Dd>, <Xn>

64-bit to top half of 128-bit (sf == 1 && type == 10 && rmode == 01 && opcode == 111)

FMOV <Vd>.D[1], <Xn>

Double-precision to 64-bit (sf == 1 && type == 01 && rmode == 00 && opcode == 110)

FMOV <Xd>, <Dn>

Top half of 128-bit to 64-bit (sf == 1 && type == 10 && rmode == 01 && opcode == 110)

FMOV <Xd>, <Vn>.D[1]
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
when '00'
  fltsize = 32;
when '01'
  fltsize = 64;
when '10'
  if opcode<2:1>:rmode != '11 01' then UNDEFINED;
  fltsize = 128;
when '11'
  if HaveFP16Ext() then
    fltsize = 16;
  else
    UNDEFINED;

case opcode<2:1>:rmode of
when '00 xx'        // FCVT[NPMZ][US]
  rounding = FPRounding(rmode);
  unsigned = (opcode<0> == '1');
  op = FPConvOp CVT_FtoI;
when '01 00'        // [US]CVTF
  rounding = FPRoundingMode(FPCR);
  unsigned = (opcode<0> == '1');
  op = FPConvOp CVT_ItoF;
when '10 00'        // FCVTA[US]
  rounding = FPRounding_TIEAWAY;
  unsigned = (opcode<0> == '1');
  op = FPConvOp CVT_FtoI;
when '11 00'        // FMOV
  if fltsize != 16 && fltsize != intsize then UNDEFINED;
  op = if opcode<0> == '1' then FPConvOp MOV_ItoF else FPConvOp MOV_FtoI;
  part = 0;
when '11 01'        // FMOV D[1]
  if intsize != 64 || fltsize != 128 then UNDEFINED;
  op = if opcode<0> == '1' then FPConvOp MOV_ItoF else FPConvOp MOV_FtoI;
  part = 1;
  fltsize = 64; // size of D[1] is 64
when '11 11'        // FJCVTZS
  if !HaveFJCVTZSExt() then UNDEFINED;
  rounding = FPRounding_ZERO;
  unsigned = (opcode<0> == '1');
  op = FPConvOp CVT_FtoI_JS;
otherwise
  UNDEFINED;

Assembler Symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<HD> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<SD> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
      fltval = V[n];
      intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
      X[d] = intval;
  when FPConvOp_CVT_ItoF
      intval = X[n];
      fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
      V[d] = fltval;
  when FPConvOp_MOV_FtoI
      fltval = Vpart[n,part];
      intval = ZeroExtend(fltval, intsize);
      X[d] = intval;
  when FPConvOp_MOV_ItoF
      intval = X[n];
      fltval = intval<fltsize-1:0>;
      Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
      fltval = V[n];
      intval = FPFixedJS(fltval, FPCR, TRUE);
      X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Half-precision (type == 11) (ARMv8.2)

FMOV <Hd>, #<imm>

Single-precision (type == 00)

FMOV <Sd>, #<imm>

Double-precision (type == 01)

FMOV <Dd>, #<imm>

integer d = UInt(Rd);

integer datasize;

case type of
when '00' datasize = 32;
when '01' datasize = 64;
when '10' UNDEFINED;
when '11'
    if HaveFP16Ext() then
dataSize = 16;
else
    UNDEFINED;
bits(datasize) imm = VFPExpandImm(imm8);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in A64 floating-point instructions.

Operation

CheckFPAdvSIMDEnabled64();

V[d] = imm;
Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register. A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  |

**Half-precision (type == 11)**

(ARMv8.2)

\[ \text{FMSUB } <\text{Hd}>, <\text{Hn}>, <\text{Hm}>, <\text{Ha}> \]

**Single-precision (type == 00)**

\[ \text{FMSUB } <\text{Sd}>, <\text{Sn}>, <\text{Sm}>, <\text{Sa}> \]

**Double-precision (type == 01)**

\[ \text{FMSUB } <\text{Dd}>, <\text{Dn}>, <\text{Dm}>, <\text{Da}> \]

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
data-size = 16;
    else
      UNDEFINED;

boolean opa_neg = (o1 == '1');
boolean opl_neg = (o0 != o1);
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Da>` Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Ha>` Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if opa_neg then operanda = FPNeg(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMUL (by element)**

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR** or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar, half-precision**, **Scalar, single-precision and double-precision**, **Vector, half-precision** and **Vector, single-precision and double-precision**.

**Scalar, half-precision**

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

**Scalar, half-precision**

```plaintext
FMUL <Hd>, <Hn>, <Vm>.H[index]
```

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
in integer datasize = esize;
in integer elements = 1;
boolean mulx_op = (U == '1');
```

**Scalar, single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | sz | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |
```

FMUL (by element)
Scalar, single-precision and double-precision

\[ \text{FMUL } <V><d>, <V><n>, <Vm>.<Ts>[<index>] \]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

Vector, half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q   | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

Vector, half-precision

\[ \text{FMUL } <Vd>, <T>, <Vn>, <T>, <Vm>.H[<index>] \]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Vector, single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q   | 0  | 0  | 1  | 1  | 1  | 1  | sz | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

FMUL (by element)
Vector, single-precision and double-precision

FMUL <Vd>, <T>, <Vn>, <T>, <Vm>, <Ts>[<index>]

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.
For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
<Ts> Is an element size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.
For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem(operand2, index, esize);

for e = 0 to elements-1
    element1 = Elem(operand1, e, esize);
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMUL (vector)

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | Rn | Rd |

**Single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | Rn | Rd |

**Assembler Symbols**

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<**T**> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;<strong>T</strong>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;<strong>T</strong>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bias(esize) element1;
bias(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR);
V[d] = result;
```
FMUL (scalar)

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 0 type 1 | Rd | 0 0 0 1 0 | Rn | Rd

op

Half-precision (type == 11)
(ARMv8.2)

FMUL <Hd>, <Hn>, <Hm>

Single-precision (type == 00)

FMUL <Sd>, <Sn>, <Sm>

Double-precision (type == 01)

FMUL <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
datasize = 16;
        else
            UNDEFINED;
boolean negated = (op == '1');
```

Assembler Symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<DN> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<DM> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<HD> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<HN> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<HM> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<SD> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<SN> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<SM> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);

if negated then result = FPNeg(result);

V[d] = result;
```
FMULX (by element)

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar, half-precision

FMULX <Hd>, <Hn>, <Vm>.H[index]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsiz = if H == 'l' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

Scalar, single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sz</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FMULX (by element)
Scalar, single-precision and double-precision

FMULX \(<V><d>, <V><n>, <Vm>.<Ts>[<index>]\)

```plaintext
integer idxdsz = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = Uint(H:L);
  when '10' index = Uint(H);
  when '11' UNDEFINED;

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rmhi:Rm);
integer esize = 32 << Uint(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Vector, half-precision

(ARMv8.2)

```
\[0\quad Q\quad 1\quad 0\quad 1\quad 1\quad 1\quad 1\quad 0\quad 0\]
\[\mathrm{L}\quad M\quad \mathrm{Rm}\quad 1\quad 0\quad 0\quad 1\quad \mathrm{H}\quad 0\]
\[\mathrm{Rn}\quad \mathrm{Rd}\]
```

Vector, half-precision

FMULX \(<Vd>, <Vn>.<T>, <Vm>.H[<index>]\)

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer idxdsz = if H == '1' then 128 else 64;
integer index = Uint(H:L:M);

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision

```
\[0\quad Q\quad 1\quad 0\quad 1\quad 1\quad 1\quad 1\quad \mathrm{sz}\quad L\quad M\quad \mathrm{Rm}\quad 1\quad 0\quad 0\quad 1\quad \mathrm{H}\quad 0\quad \mathrm{Rn}\quad \mathrm{Rd}\]
```

FMULX (by element)
Vector, single-precision and double-precision

FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

\[
\begin{array}{c|c}
\text{sz} & \text{<V>} \\
0 & S \\
1 & D \\
\end{array}
\]

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector, half-precision variant: is an arrangement specifier, encoded in "Q":

\[
\begin{array}{c|c}
\text{Q} & \text{<T>} \\
0 & 4H \\
1 & 8H \\
\end{array}
\]

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

\[
\begin{array}{c|c|c}
\text{Q} & \text{sz} & \text{<T>} \\
0 & 0 & 2S \\
0 & 1 & RESERVED \\
1 & 0 & 4S \\
1 & 1 & 2D \\
\end{array}
\]

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "sz":

\[
\begin{array}{c|c}
\text{sz} & \text{<Ts>} \\
0 & S \\
1 & D \\
\end{array}
\]

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H";
<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>H:L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsizsize) operand2 = V[m];
bits(datasize) result;
b bits( esize ) element1;
b bits( esize ) element2 = Elem[operand2, index, esize];
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  if mulx_op then
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
  else
    Elem[result, e, esize] = FPMul(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMULX**

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

### Scalar half precision

**(ARMv8.2)**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Rm | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn | 0  | 0  | 0  | 1  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

### Scalar single-precision and double-precision

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Rm | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn | 1  | 1  | 0  | 1  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

### Scalar single-precision and double-precision

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Rm | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

### Vector half precision

**(ARMv8.2)**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Rm | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn | 0  | 0  | 0  | 1  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Vector half precision

\[
\text{FMULX } <V_d>.<T>, <V_n>.<T>, <V_m>.<T>
\]

If \(!\text{HaveFP16Ext}()\) then UNDEFINED;

\[
\begin{align*}
\text{integer } d &= \text{UInt}(R_d) ; \\
\text{integer } n &= \text{UInt}(R_n) ; \\
\text{integer } m &= \text{UInt}(R_m) ; \\
\text{integer esize} &= 16 ; \\
\text{integer datasize} &= \text{if } Q == '1' \text{ then } 128 \text{ else } 64 ; \\
\text{integer elements} &= \text{datasize} \text{ DIV esize} ;
\end{align*}
\]

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 1  | 1  | 1  | Rn | Rd |

Vector single-precision and double-precision

\[
\text{FMULX } <V_d>.<T>, <V_n>.<T>, <V_m>.<T>
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(R_d) ; \\
\text{integer } n &= \text{UInt}(R_n) ; \\
\text{integer } m &= \text{UInt}(R_m) ; \\
\text{if } sz:Q == '10' \text{ then UNDEFINED} ; \\
\text{integer esize} &= 32 \ll \text{UInt}(sz) ; \\
\text{integer datasize} &= \text{if } Q == '1' \text{ then } 128 \text{ else } 64 ; \\
\text{integer elements} &= \text{datasize} \text{ DIV esize} ;
\end{align*}
\]

Assembler Symbols

\(<H_d>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<H_n>\) Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<H_m>\) Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

\(<V>\) Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<m>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<V_n>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<V_m>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNEG (vector)

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 0 | Q | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |
| U |

Half-precision

FNEG <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Single-precision and double-precision

| 0 | Q | 1 | 0 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |
| U |

Single-precision and double-precision

FNEG <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:...
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 1  | 0  | type | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | Rd | Rn |

**Half-precision (type == 11)**

(ARMv8.2)

\begin{verbatim}
FNEG <Hd>, <Hn>
\end{verbatim}

**Single-precision (type == 00)**

\begin{verbatim}
FNEG <Sd>, <Sn>
\end{verbatim}

**Double-precision (type == 01)**

\begin{verbatim}
FNEG <Dd>, <Dn>
\end{verbatim}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
\textbf{case} type of
\textbf{when} '00' datasize = 32;
\textbf{when} '01' datasize = 64;
\textbf{when} '10' UNDEFINED;
\textbf{when} '11'
    \textbf{if} HaveFP16Ext() \textbf{then}
        datasize = 16;
    \textbf{else}
        UNDEFINED;
\end{verbatim}

\begin{verbatim}
FPUnaryOp fpop;
\textbf{case} opc of
\textbf{when} '00' fpop = FPUnaryOp\_MOV;
\textbf{when} '01' fpop = FPUnaryOp\_ABS;
\textbf{when} '10' fpop = FPUnaryOp\_NEG;
\textbf{when} '11' fpop = FPUnaryOp\_SQRT;
\end{verbatim}

**Assembler Symbols**

\begin{verbatim}
<Dd> \quad \text{Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
<Dn> \quad \text{Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
<Hd> \quad \text{Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
<Hn> \quad \text{Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
<Sd> \quad \text{Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
<Sn> \quad \text{Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
\end{verbatim}
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
    when FPUnaryOp_MOV result = operand;
    when FPUnaryOp_ABS result = FPAbs(operand);
    when FPUnaryOp_NEG result = FPNeg(operand);
    when FPUnaryOp_SQRT result = FPSqrt(operand, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision (type == 11)
(ARMv8.2)

FNMADD <Hd>, <Hn>, <Hm>, <Ha>

Single-precision (type == 00)

FNMADD <Sd>, <Sn>, <Sm>, <Sa>

Double-precision (type == 01)

FNMADD <Dd>, <Dn>, <Dm>, <Da>

integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;

case type of
when '00' datasize = 32;
when '01' datasize = 64;
when '10' UNDEFINED;
when '11'
    if HaveFP16Ext () then
        datasize = 16;
    else
        UNDEFINED;

boolean opa_neg = (o1 == '1');
boolean opl_neg = (o0 != o1);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da> Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha> Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if opa_neg then operanda = FPNeg(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);
V[d] = result;
```
Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if opa_neg then operanda = FPNeg(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);
V[d] = result;
```

Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

integer d = UInt(Rd);
it integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;

case type of
when '00' datasize = 32;
when '01' datasize = 64;
when '10' UNDEFINED;
when '11'
    if HaveFP16Ext() then
        datasize = 16;
    else
        UNDEFINED;

boolean negated = (op == '1');
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
result = FPMul(operand1, operand2, FPCR);
if negated then result = FPNeg(result);
V[d] = result;
```
FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 1 0 1 1 1 1 1 0 0 1 1 1 0 1 1 1 0 1 1 0    Rn      Rd
```

Scalar single-precision and double-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 1 0 1 1 1 1 0 1 0 0 0 0 1 1 1 0 1 1 0    Rn      Rd
```

Scalar single-precision and double-precision

```
FRECPE <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32;
integer datasize = esize;
integer elements = 1;
```

Vector half precision

(ARMv8.2)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 0 0 1 1 1 1 0 1 1 1 1 1 0 0 1 1 1 0 1 1 0    Rn      Rd
```
Vector half precision

FRECPE <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

FRECPE <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPREcipEstimate(element, FPCR);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRECPS

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision.

Scalar half precision
(ARMv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0 1 0</td>
</tr>
</tbody>
</table>

Scalar half precision

FRECPS <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0 1 1</td>
</tr>
</tbody>
</table>

Scalar single-precision and double-precision

FRECPS <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision
(ARMv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 Q 0 1 1 1 0 0 1 0</td>
</tr>
</tbody>
</table>

FRECPS
Vector half precision

FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 0 0 1 1 1 0 0 sz 1 Rm 1 1 1 1 1 1 Rn Rd

Vector single-precision and double-precision

FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:  

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);
V[d] = result;
```
FRECPX

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | | Rn | Rd | |

**Single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | sz | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | | Rn | Rd | |

**Assembler Symbols**

<\(Hd\)> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<\(Hn\)> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\(V\)> Is a width specifier, encoded in "\(sz\)":

<table>
<thead>
<tr>
<th>(sz)</th>
<th>&lt;(V)&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<\(d\)> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<\(n\)> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecpX(element, FPCR);

V[d] = result;
FRINT32X (vector)

Floating-point Round to 32-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the 

FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in 

FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  |

Rn    Rd op

Vector single-precision and double-precision

FRINT32X <Vd>.<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);

V[d] = result;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00be8_rc3 ; Build timestamp: 2018-09-13T13:04
FRINT32X (scalar)

Floating-point Round to 32-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns (for the corresponding result value) the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(ARMv8.5)

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 1   | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |

<table>
<thead>
<tr>
<th>type</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Single-precision (type == 00)

FRINT32X <Sd>, <Sn>

Double-precision (type == 01)

FRINT32X <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);

Assembler Symbols

<Dr> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dr> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sr> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sr> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundIntN(operand, FPCR, rounding, intsize);

V[d] = result;
FRINT32Z (vector)

Floating-point Round to 32-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | Rn | Rd |
| U  | op |

Vector single-precision and double-precision

FRINT32Z <Vd>,<T>, <Vn>,<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);
V[d] = result;
FRINT32Z (scalar)

Floating-point Round to 32-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTER_EL2, and CPTER_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(ARMv8.5)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**type**  **op**

Single-precision (type == 00)

FRINT32Z <Sd>, <Sn>

Double-precision (type == 01)

FRINT32Z <Dd>, <Dn>

```plaintext
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
when '00' datasize = 32;
when '01' datasize = 64;
when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;Dd&gt;</th>
<th>Is the 64-bit name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Dn&gt;</td>
<td>Is the 64-bit name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;Sd&gt;</td>
<td>Is the 32-bit name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Sn&gt;</td>
<td>Is the 32-bit name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
</tbody>
</table>

Operation

```plaintext
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundIntN(operand, FPCR, rounding, intsize);
V[d] = result;
```

Page 502
FRINT64X (vector)

Floating-point Round to 64-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Vector single-precision and double-precision**

(ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |   | Rn |   | Rd |

**U**

op

**Vector single-precision and double-precision**

FRINT64X <Vd>,<T>,<Vn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);

**Asmblery Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
   element = Elem[operand, e, esize];
   Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);
V[d] = result;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
FRINT64X (scalar)

Floating-point Round to 64-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns (for the corresponding result value) the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

<table>
<thead>
<tr>
<th>type</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Single-precision (type == 00)

FRINT64X <Sd>, <Sn>

Double-precision (type == 01)

FRINT64X <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;
FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];
result = FPRoundIntN(operand, FPCR, rounding, intsize);
V[d] = result;
FRINT64Z (vector)

Floating-point Round to 64-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(ARMv8.5)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |
U | op

Vector single-precision and double-precision

FRINT64Z <Vd>..<T>, <Vn>..<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<T> Is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);
V[d] = result;
FRINT64Z (scalar)

Floating-point Round to 64-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point (ARMv8.5)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Single-precision (type == 00)**

FRINT64Z <Sd>, <Sn>

**Double-precision (type == 01)**

FRINT64Z <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;
FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundIntN(operand, FPCR, rounding, intsize);

V[d] = result;
FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 
| U  | o2 | o1 |

Half-precision

FRINTA <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext () then UNDEFINED;

integer d = Uint(Rd);
integer n = Uint(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
  case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);

### Single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Single-precision and double-precision

FRINTA \(<Vd>_.<T>, <Vn>_.<T>\)

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{if } sz:Q &= '10' \text{ then UNDEFINED; } \\
\text{integer } esize &= 32 \ll \text{UInt}(sz); \\
\text{integer } datasize &= \text{if } Q == '1' \text{ then 128 else 64}; \\
\text{integer } elements &= \text{datasize \div esize}; \\
\text{boolean } exact &= \text{FALSE}; \\
\text{FPRounding } &\text{rounding}; \\
\text{case } U:01:02 \text{ of} \\
&\text{ when } '0xx' \text{ rounding } = \text{FPDecodeRounding}(01:02); \\
&\text{ when } '100' \text{ rounding } = \text{FPRounding TIEAWAY}; \\
&\text{ when } '101' \text{ UNDEFINED; } \\
&\text{ when } '110' \text{ rounding } = \text{FPRoundingMode}(\text{FPCR}); \text{ exact } = \text{TRUE}; \\
&\text{ when } '111' \text{ rounding } = \text{FPRoundingMode}(\text{FPCR}); \\
\end{align*}
\]

Assembler Symbols

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in “Q”:

\[
\begin{array}{c|c}
Q & <T> \\
\hline
0 & 4H \\
1 & 8H \\
\end{array}
\]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

\[
\begin{array}{c|c|c|c}
sz & Q & <T> \\
\hline
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & RESERVED \\
1 & 1 & 2D \\
\end{array}
\]

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

\[
\begin{align*}
&\text{CheckFPAdvSIMDEnabled64}(); \\
&\text{bits(datasize) operand } = \text{V[n]}; \\
&\text{bits(datasize) result}; \\
&\text{bits(esize) element}; \\
&\text{for } e = 0 \text{ to elements}-1 \\
&\quad \text{element } = \text{Elem}[\text{operand}, e, \text{esize}]; \\
&\quad \text{Elem}[\text{result}, e, \text{esize}] = \text{FPRoundInt}(\text{element, FPCR, rounding, exact}); \\
&\text{V[d] } = \text{result}; \\
\end{align*}
\]
FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision (type == 11)
(ARMv8.2)

```assembly
FRINTA <Hd>, <Hn>
```

### Single-precision (type == 00)

```assembly
FRINTA <Sd>, <Sn>
```

### Double-precision (type == 01)

```assembly
FRINTA <Dd>, <Dn>
```

```python
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

### Assembler Symbols

- `<Dd>`  Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>`  Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>`  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>`  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>`  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);
V[d] = result;
```
FRINTI (vector)

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  |

U o2 o1 Rn Rd

Half-precision

FRINTI <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;

case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  |

U o2 o1 Rn Rd
Single-precision and double-precision

FRINTI <Vd>,<T>, <Vn>,<T>

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);
V[d] = result;
```
FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 1 0 type 1 0 0 1 1 1 1 1 0 0 0 0 Rn Rd</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>rmode</td>
</tr>
</tbody>
</table>
```

**Half-precision (type == 11)**

(ARMv8.2)

```
FRINTI <Hd>, <Hn>
```

**Single-precision (type == 00)**

```
FRINTI <Sd>, <Sn>
```

**Double-precision (type == 01)**

```
FRINTI <Dd>, <Dn>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;

```makefile
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;
endcase
```

boolean exact = FALSE;
FPRounding rounding;

```makefile
case rmode of
  when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
endcase
```

**Assembler Symbols**

- `<Dd>` is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```
FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | Rn | Rd |
| U  | o2 | o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Half-precision

FRINTM <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
  case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | Rn | Rd |
| U  | o2 | o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Single-precision and double-precision

FRINTM <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;

case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);
V[d] = result;
FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision (type == 11)
(ARMv8.2)

FRINTM <Hd>, <Hn>

### Single-precision (type == 00)
FRINTM <Sd>, <Sn>

### Double-precision (type == 01)

FRINTM <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
  case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
      if HaveFP16Ext() then
datasize = 16;
      else
    UNDEFINED;
  end when;

boolean exact = FALSE;
FPRounding rounding;
  case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
  end case;
```

### Assembler Symbols

- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>`: Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>`: Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>`: Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | Rd |

**Half-precision**

FRINTN <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

**Single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | Rd |

FRINTN (vector)
Single-precision and double-precision

**FRINTN <Vd>.<T>, <Vn>.<T>**

integer d = Uint(Rd);
integer n = Uint(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << Uint(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
**FPRounding** rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

**Assembler Symbols**

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

`CheckFPAvSIMDEnabled64();`
`bits(datasize) operand = V[n];`
`bits(datasize) result;`
`bits(esize) element;`

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Half-precision (type == 11)**
(ARMv8.2)

```
FRINTN <Hd>, <Hn>
```

**Single-precision (type == 00)**

```
FRINTN <Sd>, <Sn>
```

**Double-precision (type == 01)**

```
FRINTN <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
when '00' datasize = 32;
when '01' datasize = 64;
when '10' UNDEFINED;
when '11'
if HaveFP16Ext() then
datasize = 16;
else
UNDEFINED;

boolean exact = FALSE;
FPRounding rounding;
case rmode of
when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
when '100' rounding = FPRounding_TIEAWAY;
when '101' UNDEFINED;
when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
when '111' rounding = FPRoundingMode(FPCR);
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```
CheckFPAdvSIMEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```
FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(ARMv8.2)

FRINTP <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

Single-precision and double-precision

FRINTP (vector)
Single-precision and double-precision

FRINTP <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>  For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);
V[d] = result;
FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision (type == 11)  
(ARMv8.2)

```
FRINTP <Hd>, <Hn>
```

Single-precision (type == 00)

```
FRINTP <Sd>, <Sn>
```

Double-precision (type == 01)

```
FRINTP <Dd>, <Dn>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>`: Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>`: Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>`: Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMEnabled64();

bits(datasize) result;
bits(datasize) operand = \text{V}[n];

result = \text{FPRoundInt}(\text{operand}, \text{FPCR}, \text{rounding, exact});

\text{V}[d] = result;
FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When a result value is not numerically equal to the corresponding input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(ARMv8.2)

FRINTX <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

d = Data.size = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '00x' rounding = FPDecodeRounding (o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' undefined;
  when '110' rounding = FPRoundingMode (FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode (FPCR);

Single-precision and double-precision

FRINTX (vector)
Single-precision and double-precision

\[ \text{FRINTX} <V_d>, <T>, <V_n>, <T> \]

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
\end{verbatim}

Assembler Symbols

\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

\[
\begin{array}{cc}
Q & <T>\\
0 & 4H \\
1 & 8H \\
\end{array}
\]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\[
\begin{array}{ccc}
sz & Q & <T>\\
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & RESERVED \\
1 & 1 & 2D \\
\end{array}
\]

\(<V_n>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\text{bits(datasize) operand = V[n];}
\]

\[
\text{bits(datasize) result;}
\]

\[
\text{bits(esize) element;}
\]

\[
\text{for e = 0 to elements-1}
  \text{element = Elem[operand, e, esize];}
  \text{Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);}
\]

\[
V[d] = result;
\]

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When the result value is not numerically equal to the input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 1 1 0 type 1 0 0 1 1 1 0 1 0 0 0 0 Rn Rd

Half-precision (type == 11)
(ARMv8.2)

FRINTX <Hd>, <Hn>

Single-precision (type == 00)

FRINTX <Sd>, <Sn>

Double-precision (type == 01)

FRINTX <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;
  boolean exact = FALSE;
  FPRounding rounding;
case rmode of
  when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```
FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | Rn | Rd |
| U  | o2 | o1 |

Half-precision

FRINTZ <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | Rn | Rd |
| U  | o2 | o1 |
Single-precision and double-precision

PRINTZ <Vd>,<T>, <Vn>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean exact = FALSE;

FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);
V[d] = result;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 1 1 0 type 1 0 0 1 0 1 1 1 0 0 0 0 | Rd | Rn

rmode

Half-precision (type == 11)
(ARMv8.2)

FRINTZ <Hd>, <Hn>

Single-precision (type == 00)

FRINTZ <Sd>, <Sn>

Double-precision (type == 01)

FRINTZ <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext () then
      datasize = 16;
    else
      UNDEFINED;

boolean exact = FALSE;
FP Rounding rounding;
case rmode of
  when '0xx' rounding = FPDecodeRounding (rmode<1:0>);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode (FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode (FPCR);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```
FRSQRTE

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar half precision

FRSQRTE <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar single-precision and double-precision

FRSQRTE <V>d>, <V>n>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision
(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>O</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector half precision

FRSQRTE <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

FRSQRTE <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR);
V[d] = result;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision.

### Scalar half precision (ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | Rm |

### Scalar half precision

FSQRTS `<Hd>`, `<Hn>`, `<Hm>`

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

### Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | sz | 1  | Rm |

### Scalar single-precision and double-precision

FSQRTS `<V><d>`, `<V><n>`, `<V><m>`

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

### Vector half precision (ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | Rm |

### Vector half precision

integeresize = 16;
integer datasize = esize;
integer elements = 1;
Vector half precision

FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 0  | 1  | sz | 1  | Rm | 1  | 1  | 1  | 1  | 1  | 1  | 1  | Rn | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | Rd |

Vector single-precision and double-precision

FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Hz> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);
V[d] = result;
```
FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | Rn |

<table>
<thead>
<tr>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Half-precision

FSQRT <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | Rn |

<table>
<thead>
<tr>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Single-precision and double-precision

FSQRT <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-008rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  |   | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |

opc

Half-precision (type == 11)
(ARMv8.2)

FSQRT <Hd>, <Hn>

Single-precision (type == 00)

FSQRT <Sd>, <Sn>

Double-precision (type == 01)

FSQRT <Dd>, <Dn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFPL6Ext() then
            datasize = 16;
        else
            UNDEFINED;
    case opc of
        when '00' fpop = FPUnaryOp_MOV;
        when '01' fpop = FPUnaryOp_ABS;
        when '10' fpop = FPUnaryOp_NEG;
        when '11' fpop = FPUnaryOp_SQRT;
```

Assembler Symbols

- `<Dd>` is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
void CheckFPAdvSIMDEnabled64();

void fpop();

void CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

void fpop();

void CheckFPAdvSIMDEnabled64();

case fpop of
    when FPUnaryOp_MOV result = operand;
    when FPUnaryOp_ABS result = FPAbs(operand);
    when FPUnaryOp_NEG result = FPNeg(operand);
    when FPUnaryOp_SQRT result = FPSqrt(operand, FPCR);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSUB (vector)

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPSR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 1  | 0  | Rm | 0  | 0  | 0  | 1  | 0  | Rd |

**Single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 1  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 1  | 0  | Rd |

**Assembler Symbols**

<\text{T}> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>0</th>
<th>4H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:
### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) diff;

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  diff = FPSub(element1, element2, FPCR);
  Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```
FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 0  | 0  | 0  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| op |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Half-precision (type == 11)
(ARMv8.2)

FSUB <Hd>, <Hn>, <Hm>

Single-precision (type == 00)

FSUB <Sd>, <Sn>, <Sm>

Double-precision (type == 01)

FSUB <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
data size = 16;
        else
            UNDEFINED;
        end

boolean sub_op = (op == '1');

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if sub_op then
    result = FPSub(operand1, operand2, FPCR);
else
    result = FPAdd(operand1, operand2, FPCR);

V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

**Integer (ARMv8.5)**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>&quot;Xm&quot;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>&quot;Xn&quot;</td>
<td>&quot;Xd&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Integer**

GMI \(<Xd>\), \(<Xn|SP>\), \(<Xm>\)

```plaintext
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

**Assembler Symbols**

\(<Xd>\)  
Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.

\(<Xn|SP>\)  
Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

\(<Xm>\)  
Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field.

**Operation**

```plaintext
bits(64) address = if n == 31 then SP[] else X[n];
bits(64) mask = X[m];
bits(4) tag = AllocationTagFromAddress(address);
mask<UInt(tag)> = '1';
X[d] = mask;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions. Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

System

HINT #<imm>

SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 111' op = System HintOp_YIELD; // SEE "XPACLRI"
    when '0001 xxx' op = SystemHintOp_YIELD; // SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716"
    when '0010 000' if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_ESE;
    when '0010 001' if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_PSB;
    when '0010 010' if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_TSB;
    when '0010 100' op = SystemHintOp_CSDB;
    when '0011 xxx' op = SystemHintOp_CSDB;
    when '0100 xx0' op = SystemHintOp_BT1;
        BTypeCompatible = BTypeCompatible_BT1(op2<2:1>);
    otherwise EndOfInstruction(); // Instruction executes as NOP

Assembler Symbols

<imm> is a 7-bit unsigned immediate, in the range 0 to 127 encoded in the "CRm:op2" field.

The encodings that are allocated to architectural hint functionality are described in the "Hints" table in the "Index by Encoding".

For allocated encodings of "CRm:op2":
- A disassembler will disassemble the allocated instruction, rather than the HINT instruction.
- An assembler may support assembly of allocated encodings using HINT with the corresponding <imm> value, but it is not required to do so.
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, TRUE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, TRUE);
      WaitForEvent();
  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap(EL1, FALSE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap(EL3, FALSE);
      WaitForInterrupt();
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
      AArch64.vESBOperation();
      TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSD
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    BTypeNext = '00';
  otherwise // do nothing
HLT

Halt instruction. An HLT instruction can generate a Halt Instruction debug event, which causes entry into Debug state.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | imm16 | 0  | 0  | 0  | 0  |

System

HLT #<imm>

if EDSR.HDE == '0' || !HaltingAllowed() then UndefinedFault();
if HaveBTIExt() then
  BTypeCompatible = TRUE;

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

Halt(DebugHalt_HaltInstruction);
HVC

Hypervisor Call causes an exception to EL2. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- At EL0, and Secure EL1.
- When `SCR_EL3.HCE` is set to 0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in `ESR_ELx`, using the EC value 0x16, and the value of the immediate argument.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

System

HVC #<imm>

```
bits(16) imm = imm16;
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && (!IsSecureEL2Enabled() && IsSecure())) then
  UNDEFINED;

hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);
if hvc_enable == '0' then
  AArch64.UndefinedFault();
else
  AArch64.CallHypervisor(imm);
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
IC

Instruction Cache operation. For more information, see .

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>op1</th>
<th>0 1 1 1</th>
<th>CRm</th>
<th>op2</th>
<th>Rt</th>
</tr>
</thead>
</table>
| L | CRn

System

IC <ic_op>, <Xt>

is equivalent to

SYS #<op1>, C7, <Cm>, #<op2>, <Xt>

and is the preferred disassembly when SysOp(op1,'0111',CRm,op2) == Sys_IC.

Assembler Symbols

<ic_op> Is an IC instruction name, as listed for the IC system instruction pages, encoded in “op1:CRm:op2”:

<table>
<thead>
<tr>
<th>op1</th>
<th>CRm</th>
<th>op2</th>
<th>&lt;ic_op&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0001</td>
<td>000</td>
<td>IALLUIS</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>000</td>
<td>IALLU</td>
</tr>
<tr>
<td>011</td>
<td>0101</td>
<td>001</td>
<td>IVAU</td>
</tr>
</tbody>
</table>

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.
INS (element)

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (element).

### Advanced SIMD

\[
\text{INS } <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>] 
\]

integer \( d = \text{UInt}(\text{Rd}); \)
integer \( n = \text{UInt}(\text{Rn}); \)
integer size = \text{LowestSetBit}(\text{imm5});
if size > 3 then UNDEFINED;
integer dst_index = \text{UInt}(\text{imm5}<4:\text{size}+1>);
integer src_index = \text{UInt}(\text{imm4}<3:\text{size}>)
integer idxdsizex = if imm4<3> == '1' then 128 else 64;
// imm4<size-1:0> is IGNORED
integer esize = 8 << size;

### Assembler Symbols

<\text{Vd}> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<\text{Ts}> Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>\text{imm5}</th>
<th>\text{Ts}</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx01</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<\text{index1}> Is the destination element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>\text{imm5}</th>
<th>\text{index1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx01</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

<\text{Vn}> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<\text{index2}> Is the source element index encoded in “imm5:imm4”:

<table>
<thead>
<tr>
<th>\text{imm5}</th>
<th>\text{index2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx01</td>
<td>imm4&lt;3:0&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm4&lt;3:1&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm4&lt;3:2&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm4&lt;3&gt;</td>
</tr>
</tbody>
</table>

Unspecified bits in “imm4” are ignored but should be set to zero by an assembler.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsz) operand = V[n];
bits(128) result;

result = V[d];
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
INS (general)

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (from general).

Advanced SIMD

INS <Vd>.<Ts>[<index>], <R><n>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = 128;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ts> Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> Is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>W</td>
</tr>
<tr>
<td>xxxx10</td>
<td>W</td>
</tr>
<tr>
<td>xx100</td>
<td>W</td>
</tr>
<tr>
<td>x1000</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;

result = Y[d];
Elem[result, index, esize] = element;
Y[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

**Integer**

(ARMv8.5)

|   | 75 | 74 | 73 | 72 | 71 | 70 | 69 | 68 | 67 | 66 | 65 | 64 | 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 |
| 31 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | Xm | 0 | 0 | 0 | 1 | 0 | 0 | Xn | Xd |

**Assembler Symbols**

- `<Xd|SP>` is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
- `<Xn|SP>` is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<Xm>` is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

**Operation**

```plaintext
bits(64) operand = if n == 31 then SP[] else X[n];
bits(64) xm = X[m];
bits(16) exclude = xm<15:0> OR GCR_EL1.Exclude;
if AllocationTagAccessIsEnabled() then
    if GCR_EL1.RRND == '1' then
        Rtag = _ChooseRandomNonExcludedTag(exclude);% Choose a random non-excluded tag
    else
        bits(4) start = RGSR_EL1.TAG;
bits(4) offset = RandomTag();
        Rtag = ChooseNonExcludedTag(start, offset, exclude);
    RGSR_EL1.TAG = rtag;
else
    rtag = '0000';

bits(64) result = AddressWithAllocationTag(operand, rtag);
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```
Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)*.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 1 0 1 1 1 1 CRm | opc
```

**System**

ISB `{|<option>|<imm>}`

```
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if HaveSBExt() && CRm[3:0] == '0000' then
      op = MemBarrierOp_SB;
    else
      UNDEFINED;

case CRm[3:2] of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

case CRm[1:0] of
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
  otherwise
    if CRm[3:2] == '01' then
      op = MemBarrierOp_PSSBB;
    elsif CRm[3:2] == '00' && opc == '00' then
      op = MemBarrierOp_SSBB;
    elsif HaveSBExt() && CRm[3:2] == '00' && opc == '11' then
      op = MemBarrierOp_SB;
    else
      types = MBReqTypes_All;
      domain = MBReqDomain_FullSystem;
```

**Assembler Symbols**

*<option>* Specifies an optional limitation on the barrier operation. Values are:

- **SY** Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.

All other encodings of CRm are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

*<imm>* Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.
case op of
    when MemBarrierOp_DSB
        DataSynchronizationBarrier(domain, types);
    when MemBarrierOp_DMB
        DataMemoryBarrier(domain, types);
    when MemBarrierOp_ISB
        InstructionSynchronizationBarrier();
    when MemBarrierOp_SSBB
        SpeculativeSynchronizationBarrierToVA();
    when MemBarrierOp_PSSBB
        SpeculativeSynchronizationBarrierToPA();
    when MemBarrierOp_SB
        SpeculationBarrier();
LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | x  | size | Rn |    |    |

L

opcode

One register (opcode == 0111)

LD1 { <Vt>.<T> }, [<Xn|SP>]

Two registers (opcode == 1010)

LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

Three registers (opcode == 0110)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

Four registers (opcode == 0010)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | Rm | x  | x  | 1  | x  | size | Rn |    |    |

L

opcode
One register, immediate offset (Rm == 1111 && opcode == 0111)

LD1 { <Vt>.<T> }, [Xn|SP], <imm>

One register, register offset (Rm != 1111 && opcode == 0111)

LD1 { <Vt>.<T> }, [Xn|SP], <Xm>

Two registers, immediate offset (Rm == 1111 && opcode == 1010)

LD1 { <Vt>.<T>, <Vt2>.<T> }, [Xn|SP], <imm>

Two registers, register offset (Rm != 1111 && opcode == 1010)

LD1 { <Vt>.<T>, <Vt2>.<T> }, [Xn|SP], <Xm>

Three registers, immediate offset (Rm == 1111 && opcode == 0110)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP], <imm>

Three registers, register offset (Rm != 1111 && opcode == 0110)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP], <Xm>

Four registers, immediate offset (Rm == 1111 && opcode == 0010)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [Xn|SP], <imm>

Four registers, register offset (Rm != 1111 && opcode == 0010)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [Xn|SP], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in “Q”:
For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#8</td>
</tr>
<tr>
<td>1</td>
<td>#16</td>
</tr>
</tbody>
</table>

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```plaintext
Shared Decode

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bite(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \textit{No offset} and \textit{Post-index}.

\textbf{No offset}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | x  | x  | 0  | S  | size | Rn | Rt |
| L  | R  | opcode |

8-bit (opcode == 000)

\texttt{LD1 (Vt) .B}[<index>], [<Xn|SP>]

16-bit (opcode == 010 && size == x0)

\texttt{LD1 (Vt) .H}[<index>], [<Xn|SP>]

32-bit (opcode == 100 && size == 00)

\texttt{LD1 (Vt) .S}[<index>], [<Xn|SP>]

64-bit (opcode == 100 && S == 0 && size == 01)

\texttt{LD1 (Vt) .D}[<index>], [<Xn|SP>]

\begin{verbatim}
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
\end{verbatim}

\textbf{Post-index}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | x  | x  | 0  | S  | size | Rn | Rt |
| L  | R  | opcode |
8-bit, immediate offset (Rm == 11111 && opcode == 000)

    LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1

8-bit, register offset (Rm != 11111 && opcode == 000)

    LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

    LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

    LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == x0)

    LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4

32-bit, register offset (Rm != 11111 && opcode == 100 && size == x0)

    LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == x1)

    LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == x1)

    LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>

integer t = Uint(Rt);
integer n = Uint(Rn);
integer m = Uint(Rm);
boolean wback = TRUE;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;
    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bites(64) offs;
bits(128) rval;
bites(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

**No offset**

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>R</td>
<td>size</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Post-index**

```
|   | 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | R | opcode | S | Rm | size | Rn | Rt |
```

Immediate offset (Rm == 1111)

```
LD1R { <Vt>.<T> }, [Xn|SP], <imm>
```

Register offset (Rm != 1111)

```
LD1R { <Vt>.<T> }, [Xn|SP], <Xm>
```

**Assembler Symbols**

- `<V>` is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the post-index immediate offset, encoded in "size":

Page 575
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#1</td>
</tr>
<tr>
<td>01</td>
<td>#2</td>
</tr>
<tr>
<td>10</td>
<td>#4</td>
</tr>
<tr>
<td>11</td>
<td>#8</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
integer scale = UInt(opcode<2:1>);
integer selm = UInt(opcode<0>R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(QS:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(QS:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(QS);  // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);  // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 0 0 1 1 0 0 0 0 0</td>
<td>size</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

LD2 {<Vt>.<T>, <Vt2>.<T>, [<Xn|SP>]}

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rm</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 1 1 0</td>
<td>size</td>
<td>1 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

LD2 {<Vt>.<T>, <Vt2>.<T>, [<Xn|SP>]}, <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

Immediate offset (Rm == 11111)

LD2 {<Vt>.<T>, <Vt2>.<T>, [<Xn|SP>]}, <imm>

Register offset (Rm != 11111)

LD2 {<Vt>.<T>, <Vt2>.<T>, [<Xn|SP>]}, <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp LOAD else MemOp STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

LD2 (multiple structures)
Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD2 (single structure)

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to
the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.
Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an
attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

8-bit (opcode == 000)

LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]

16-bit (opcode == 010 && size == x0)

LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]

32-bit (opcode == 100 && size == 00)

LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]

64-bit (opcode == 100 && S == 0 && size == 01)

LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]

Post-index

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
8-bit, immediate offset (Rm == 11111 && opcode == 000)

LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2

8-bit, register offset (Rm != 11111 && opcode == 000)

LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
:index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
      scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if \texttt{HaveMTEExt}() then
  \texttt{SetNotTagCheckedInstruction(!wback && n == 31);}

\texttt{CheckFPAdvSIMDEnabled64();}

\texttt{bits(64) address; bits(64) offs; bits(128) rval; bits(esize) element;}
constant integer ebytes = esize DIV 8;

if n == 31 then
  \texttt{CheckSPAlignment();}
  \texttt{address = SP[];}
else
  \texttt{address = X[n];}

\texttt{offs = zeros();}
if replicate then
  \begin{verbatim}
  // load and replicate to all elements
  for s = 0 to selem-1
    element = \texttt{Mem[address + offs, ebytes, AccType_VEC]};
    // replicate to fill 128- or 64-bit register
    \texttt{V[t] = Replicate(element, datasize DIV esize);}
    \texttt{offs = offs + ebytes;}
    \texttt{t = (t + 1) MOD 32;}
  \end{verbatim}
else
  \begin{verbatim}
  // load/store one element per register
  for s = 0 to selem-1
    rval = \texttt{V[t];}
    if memop == MemOp\_LOAD then
      // insert into one lane of 128-bit register
      \texttt{Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType\_VEC];}
      \texttt{V[t] = rval;}
    else // memop == MemOp\_STORE
      // extract from one lane of 128-bit register
      \texttt{Mem[address + offs, ebytes, AccType\_VEC] = Elem[rval, index, esize];}
    \texttt{offs = offs + ebytes;}
    \texttt{t = (t + 1) MOD 32;}
  \end{verbatim}
if wback then
  if m != 31 then
    \texttt{offs = X[m];}
  if n == 31 then
    \texttt{SP[] = address + offs;}
else
  \texttt{X[n] = address + offs;}

\begin{quote}
\textbf{Operational information}
\end{quote}

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers. Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{No offset} and \texttt{Post-index}.

### No offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

L R opcode S

### Post-index

Immediate offset (Rm == 11111)

LD2R (\texttt{<Vt>.<T>, <Vt2>.<T> }, [\texttt{<Xn|SP>}], \texttt{<imm>})

Register offset (Rm != 11111)

LD2R (\texttt{<Vt>.<T>, <Vt2>.<T> }, [\texttt{<Xn|SP>}], \texttt{<Xm>})

\begin{verbatim}
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
\end{verbatim}

### Assembler Symbols

\texttt{<Vt>} Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

\texttt{<T>} Is an arrangement specifier, encoded in "size:Q":

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
 size & Q & <T> \\
\hline
 00 & 0 & 8B \\
 00 & 1 & 16B \\
 01 & 0 & 4H \\
 01 & 1 & 8H \\
 10 & 0 & 2S \\
 10 & 1 & 4S \\
 11 & 0 & 1D \\
 11 & 1 & 2D \\
\hline
\end{tabular}
\end{table}

\texttt{<Vt2>} Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

\texttt{<Xn|SP>} Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Is the post-index immediate offset, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#2</td>
</tr>
<tr>
<td>01</td>
<td>#4</td>
</tr>
<tr>
<td>10</td>
<td>#8</td>
</tr>
<tr>
<td>11</td>
<td>#16</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);        // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);          // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD3 (multiple structures)

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:

A is a packed array of 3-element structures. Each element is a 16-bit halfword.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |
| Q  | size | Rn | Rt |

| opcode |

No offset

LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | Rm | 0  | 1  | 0  | 0  | size | Rn | Rt |
| Q  |          |          |          |          |          |          |          |          |          |          |          |          |          |          |

| opcode |

Immediate offset (Rm == 11111)

LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
Assembler Symbols

\(<V_t>\) Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

\(<T>\) Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<V_t2>\) Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

\(<V_t3>\) Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

\(<X_n|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{imm}>\) Is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;\text{imm}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

\(<X_m>\) Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bite(64) offs;
bite(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD3 (single structure)

Load single 3-element structure to one lane of three registers. This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index.

No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn | Rt |

L  R  opcode

8-bit (opcode == 001)


16-bit (opcode == 011 && size == x0)

LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]

32-bit (opcode == 101 && size == 00)


64-bit (opcode == 101 && S == 0 && size == 01)

LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | x  | x  | 1  | S  | size | Rn | Rt |

L  R  opcode
8-bit, immediate offset (Rm == 11111 && opcode == 001)

LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3

8-bit, register offset (Rm != 11111 && opcode == 001)

LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

**Assembler Symbols**

- `<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- `<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- `<index>` For the 8-bit variant: is the element index, encoded in "Q:S:size".
  For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
  For the 32-bit variant: is the element index, encoded in "Q:S".
  For the 64-bit variant: is the element index, encoded in "Q".
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>R) + 1;
boolean replicate = FALSE;
integer index;
case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);     // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);   // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);         // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);           // D[0-1]
    scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
    if memop == MemOp_LOAD then
        // insert into one lane of 128-bit register
        Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
        V[t] = rval;
    else  // memop == MemOp_STORE
        // extract from one lane of 128-bit register
        Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

opcode  S

| Rm | 1  | 1  | 1  | 1  | size | Rn | Rt |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| L  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Immediate offset (Rm == 11111)

LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP], <imm>

Register offset (Rm != 11111)

LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP], <Xm>

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Is the post-index immediate offset, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#3</td>
</tr>
<tr>
<td>01</td>
<td>#6</td>
</tr>
<tr>
<td>10</td>
<td>#12</td>
</tr>
<tr>
<td>11</td>
<td>#24</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
to-integer scale = UInt(opcode<2:1>);
to-integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
to-integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);         // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);      // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);            // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);              // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
to-integer datasize = if Q == '1' then 128 else 64;
to-integer esize = 8 << scale;
```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4 (multiple structures)

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTNR_EL2, and CPTNR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No offset


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Immediate offset (Rm == 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```java
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;  // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;   // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;   // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;   // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;   // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 3;   // LD/ST3 (3 registers)
  when '1000' rpt = 1; selem = 2;   // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;   // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \textit{No offset} and \textit{Post-index}.

\textbf{No offset}

\begin{verbatim}
8-bit (opcode == 001)
\end{verbatim}

\begin{verbatim}
16-bit (opcode == 011 \&\& \text{size} == x0)
\end{verbatim}

\begin{verbatim}
32-bit (opcode == 101 \&\& \text{size} == 00)
\end{verbatim}

\begin{verbatim}
64-bit (opcode == 101 \&\& \text{S} == 0 \&\& \text{size} == 01)
\end{verbatim}

\begin{verbatim}
integer t = \texttt{UInt}(Rt);
integer n = \texttt{UInt}(Rn);
integer m = integer \texttt{UNKNOWN};
boolean wback = FALSE;
\end{verbatim}

\textbf{Post-index}

\begin{verbatim}
32-bit (opcode == 101 \&\& \text{size} == 11)
\end{verbatim}

\begin{verbatim}
integer t = \texttt{Int}(Rt);
integer n = \texttt{Int}(Rn);
integer m = integer \texttt{UNKNOWN};

boolean wback = FALSE;
\end{verbatim}
8-bit, immediate offset (Rm == 11111 && opcode == 001)

\[
\text{LD4} \{ \langle Vt \rangle, \langle Vt2 \rangle, \langle Vt3 \rangle, \langle Vt4 \rangle \} [\langle \text{index} \rangle], [\langle Xn|SP \rangle], \#4
\]

8-bit, register offset (Rm != 11111 && opcode == 001)

\[
\text{LD4} \{ \langle Vt \rangle, \langle Vt2 \rangle, \langle Vt3 \rangle, \langle Vt4 \rangle \} [\langle \text{index} \rangle], [\langle Xn|SP \rangle], \langle Xm \rangle
\]

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

\[
\text{LD4} \{ \langle Vt \rangle, \langle Vt2 \rangle, \langle Vt3 \rangle, \langle Vt4 \rangle \} [\langle \text{index} \rangle], [\langle Xn|SP \rangle], \#8
\]

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

\[
\text{LD4} \{ \langle Vt \rangle, \langle Vt2 \rangle, \langle Vt3 \rangle, \langle Vt4 \rangle \} [\langle \text{index} \rangle], [\langle Xn|SP \rangle], \langle Xm \rangle
\]

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

\[
\text{LD4} \{ \langle Vt \rangle, \langle Vt2 \rangle, \langle Vt3 \rangle, \langle Vt4 \rangle \} [\langle \text{index} \rangle], [\langle Xn|SP \rangle], \#16
\]

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

\[
\text{LD4} \{ \langle Vt \rangle, \langle Vt2 \rangle, \langle Vt3 \rangle, \langle Vt4 \rangle \} [\langle \text{index} \rangle], [\langle Xn|SP \rangle], \langle Xm \rangle
\]

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

\[
\text{LD4} \{ \langle Vt \rangle, \langle Vt2 \rangle, \langle Vt3 \rangle, \langle Vt4 \rangle \} [\langle \text{index} \rangle], [\langle Xn|SP \rangle], \#32
\]

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

\[
\text{LD4} \{ \langle Vt \rangle, \langle Vt2 \rangle, \langle Vt3 \rangle, \langle Vt4 \rangle \} [\langle \text{index} \rangle], [\langle Xn|SP \rangle], \langle Xm \rangle
\]

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

**Assembler Symbols**

- `<Vt>`
  - Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

- `<Vt2>`
  - Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- `<Vt3>`
  - Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

- `<Vt4>`
  - Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

- `<index>`
  - For the 8-bit variant: is the element index, encoded in "Q:S:size".
  - For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
  - For the 32-bit variant: is the element index, encoded in "Q:S".
  - For the 64-bit variant: is the element index, encoded in "Q".

- `<Xn|SP>`
  - Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- `<Xm>`
  - Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer scale = \texttt{UInt}(\text{opcode}<2:1>);
integer selem = \texttt{UInt}(\text{opcode}<0>:R) + 1;
boolean replicate = \text{FALSE};
integer index;

case scale of
   when 3
      // load and replicate
      if L == '0' || S == '1' then UNDEFINED;
      scale = \texttt{UInt}(\text{size});
      replicate = \text{TRUE};
   when 0
      index = \texttt{UInt}(Q:S:}\text{size}; // B[0-15]
   when 1
      if size<0> == '1' then UNDEFINED;
      index = \texttt{UInt}(Q:S:}\text{size<1>}; // H[0-7]
   when 2
      if size<1> == '1' then UNDEFINED;
      if size<0> == '0' then
         index = \texttt{UInt}(Q:S); // S[0-3]
      else
         if S == '1' then UNDEFINED;
         index = \texttt{UInt}(Q); // D[0-1]
         scale = 3;
     \end{cases}

\texttt{MemOp} memop = if L == '1' then \texttt{MemOp LOAD} else \texttt{MemOp STORE};
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 \ll scale;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index.

**No offset**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q| O  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  |    | size | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   | L  | R  |   |   |   |   |    |   |   |   |   |   |    | opcode |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| L | R  |   |   |   |   |   |    |   |   |   |   |   |    |     |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**No offset**


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

**Post-index**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q| O  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  |    | size | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   | L  | R  |   |   |   |   |    |   |   |   |   |   |    | opcode |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| L | R  |   |   |   |   |   |    |   |   |   |   |   |    |     |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Immediate offset (Rm == 11111)**


**Register offset (Rm != 11111)**

LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

**Assembler Symbols**

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Is the post-index immediate offset, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#4</td>
</tr>
<tr>
<td>01</td>
<td>#8</td>
</tr>
<tr>
<td>10</td>
<td>#16</td>
</tr>
<tr>
<td>11</td>
<td>#32</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```java
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);     // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);         // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);           // D[0-1]
      scale = 3;
    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
    integer datasize = if Q == '1' then 128 else 64;
    integer esize = 8 << scale;
```

LD4R
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = Zeros();

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDADDA and LDADDAL load from memory with acquire semantics.
- LDADDL and LDADDAL store to memory with release semantics.
- LDADD has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STADD, STADDL.

Integer
(ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
32-bit LDADD (size == 10 && A == 0 && R == 0)

LDADD <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDA (size == 10 && A == 1 && R == 0)

LDADDA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDAL (size == 10 && A == 1 && R == 1)

LDADDAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDL (size == 10 && A == 0 && R == 1)

LDADDL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDADD (size == 11 && A == 0 && R == 0)

LDADD <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDA (size == 11 && A == 1 && R == 0)

LDADDA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDAL (size == 11 && A == 1 && R == 1)

LDADDAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL (size == 11 && A == 0 && R == 1)

LDADDL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADD, STADDL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD  result = data + value;
    when MemAtomicOp_BIC  result = data AND NOT(value);
    when MemAtomicOp_EOR  result = data EOR value;
    when MemAtomicOp_ORR  result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDADDB, LDADDAB, LDADDALB, LDADDLB**

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, **LDADDAB** and **LDADDALB** load from memory with acquire semantics.
- **LDADDLB** and **LDADDALB** store to memory with release semantics.
- **LDADDB** has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias **STADDB, STADDLB**.

**Integer (ARMv8.1)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 0  | 0  | 0  | Rn | Rt |

**size**  **opc**

**LDADDB (A == 1 & & R == 0)**

LDADDB <Ws>, <Wt>, [<Xn|SP]>

**LDADDALB (A == 1 & & R == 1)**

LDADDALB <Ws>, <Wt>, [<Xn|SP]>

**LDADDB (A == 0 & & R == 0)**

LDADDB <Ws>, <Wt>, [<Xn|SP]>

**LDADDLB (A == 0 & & R == 1)**

LDADDLB <Ws>, <Wt>, [<Xn|SP]>

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADDB, STADDLB</td>
<td>( A = '0' &amp;&amp; \text{Rt} = '11111' )</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

  case op of
    when MemAtomicOp_ADD   result = data + value;
    when MemAtomicOp_BIC   result = data \&\& \neg(value);
    when MemAtomicOp_EOR   result = data \oplus value;
    when MemAtomicOp_ORR   result = data OR value;
    when MemAtomicOp_SMAX  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN  result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX  result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN  result = if UInt(data) > UInt(value) then value else data;

  Mem[address, datasize DIV 8, stacctype] = result;

  if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational Information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

• If the destination register is not WZR, LDADDAH and LDADDALH load from memory with acquire semantics.
• LDADDLH and LDADDALH store to memory with release semantics.
• LDADDH has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STADDDH, STADDDLH.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

size opc

LDADDAH (A == 1 && R == 0)

LDADDAH <Ws>, <Wt>, [<Xn|SP>]

LDADDALH (A == 1 && R == 1)

LDADDALH <Ws>, <Wt>, [<Xn|SP>]

LDADDH (A == 0 && R == 0)

LDADDH <Ws>, <Wt>, [<Xn|SP>]

LDADDLH (A == 0 && R == 1)

LDADDLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADDH, STADDLH</td>
<td>( A = '0' &amp;&amp; Rt = '11111' )</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
b境外(datasize) value;
b境外(datasize) data;
b境外(datasize) result;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);
value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
Mem[address, datasize DIV 8, stacctype] = result;
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

• There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release followed by a Load-AcquirePC instruction.

• The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

Integer

(ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | (1)| (1)| (1)| (1)| 1  | 1  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |

size | Rs

32-bit (size == 10)

LDAPR <Wt>, [<Xn|SP> (,#0)]

64-bit (size == 11)

LDAPR <Xt>, [<Xn|SP> (,#0)]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = AccType_ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPRB

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

### Assembler Symbols

<\textit{Wt}>\quad Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<\textit{Xn}|SP>\quad Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```plaintext
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

Integer

(ARMv8.3)

```
 integer n = UInt(Rn);
 integer t = UInt(Rt);
 integer s = UInt(Rs); // ignored by all loads and store-release
```

AccType acctype = AccType_ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;

Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (size == 10)

LDAPUR <Wt>, [<Xn|SP>{, #<simm>}

64-bit (size == 11)

LDAPUR <Xt>, [<Xn|SP>{, #<simm>}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bite(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
if \texttt{HaveMTEExt}() then
\begin{verbatim}
  boolean is_load_store = memop \in \{\texttt{MemOp\_STORE}, \texttt{MemOp\_LOAD}\};
  SetNotTagCheckedInstruction(is_load_store \&\& n == 31 \&\& \neg \texttt{wback});
\end{verbatim}

\begin{verbatim}
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
\end{verbatim}

if memop == \texttt{MemOp\_LOAD} \&\& \texttt{wback} \&\& n == t \&\& n != 31 then
\begin{verbatim}
c = \texttt{ConstrainUnpredictable}(\texttt{Unpredictable\_WBOVERLAPLD});
assert c \in \{\texttt{Constraint\_WBSUPPRESS}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDEF}, \texttt{Constraint\_NOP}\};
case c of
  when \texttt{Constraint\_WBSUPPRESS} wback = FALSE; \quad \text{// writeback is suppressed}
  when \texttt{Constraint\_UNKNOWN} \quad \text{wb\_unknown = TRUE; \quad \text{// writeback is UNKNOWN}}
  when \texttt{Constraint\_UNDEF} \quad \text{UNDEFINED;}
  when \texttt{Constraint\_NOP} \quad \text{EndOfInstruction();}
\end{verbatim}

if memop == \texttt{MemOp\_STORE} \&\& \texttt{wback} \&\& n == t \&\& n != 31 then
\begin{verbatim}
c = \texttt{ConstrainUnpredictable}(\texttt{Unpredictable\_WBOVERLAPST});
assert c \in \{\texttt{Constraint\_NONE}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDEF}, \texttt{Constraint\_NOP}\};
case c of
  when \texttt{Constraint\_NONE} rt\_unknown = FALSE; \quad \text{// value stored is original value}
  when \texttt{Constraint\_UNKNOWN} \quad \text{rt\_unknown = TRUE; \quad \text{// value stored is UNKNOWN}}
  when \texttt{Constraint\_UNDEF} \quad \text{UNDEFINED;}
  when \texttt{Constraint\_NOP} \quad \text{EndOfInstruction();}
\end{verbatim}

if n == 31 then
\begin{verbatim}
if memop != \texttt{MemOp\_PREFETCH} then \texttt{CheckSPAlignment}();
  address = SP[\ ];
else
  address = X[n];
if !\texttt{postindex then}
  address = address + offset;
\end{verbatim}

\begin{verbatim}
case memop of
  when \texttt{MemOp\_STORE}
    if rt\_unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
      \texttt{Mem}[address, datasize DIV 8, acctype] = data;
  when \texttt{MemOp\_LOAD}
    data = \texttt{Mem}[address, datasize DIV 8, acctype];
    if signed then
      X[t] = \texttt{SignExtend}(data, regsize);
    else
      X[t] = \texttt{ZeroExtend}(data, regsize);
  when \texttt{MemOp\_PREFETCH}
    \texttt{Prefetch}(address, t<4:0>);
if wback then
\end{verbatim}

\begin{verbatim}
if wb\_unknown then
  address = bits(64) UNKNOWN;
elsif postindex then
  address = address + offset;
if n == 31 then
  SP[\ ] = address;
else
  X[n] = address;
\end{verbatim}
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load-Acquire RCpc Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

### Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`: Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType accctype = AccType_ORDERED;
MemOp memop;
boolean signed;
ingtem regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURH

Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
// store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    bits(64) address;
    bits(datasize) data;

    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
        end case;

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
        end case;

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[0];
    else
        address = X[n];
    end if

    if ! postindex then
        address = address + offset;
    end if

    case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
            end if
        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
            end if
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);
        end case

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        end if
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
        end if
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURSB

Load-Acquire RCpc Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

### Assembler Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Wt&gt;</td>
<td>Is the 32-bit name of the general-purpose register to be transferred, encoded in the &quot;Rt&quot; field.</td>
</tr>
<tr>
<td>&lt;Xt&gt;</td>
<td>Is the 64-bit name of the general-purpose register to be transferred, encoded in the &quot;Rt&quot; field.</td>
</tr>
<tr>
<td>&lt;Xn</td>
<td>SP&gt;</td>
</tr>
<tr>
<td>&lt;simm&gt;</td>
<td>Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the &quot;imm9&quot; field.</td>
</tr>
</tbody>
</table>
integer n = \texttt{UInt}(Rn);
integer t = \texttt{UInt}(Rt);
\texttt{AccType} accType = \texttt{AccType\ ORDERED};
\texttt{MemOp} memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then \texttt{MemOp\ LOAD} else \texttt{MemOp\ STORE};
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = \texttt{MemOp\ PREFETCH};
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = \texttt{MemOp\ LOAD};
    if size == '10' \&\& opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
Operation

if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURSH

Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

32-bit (opc == 11)

LDAPURSH <Wt>, [<Xn|SP>{, #<simm>}]}

64-bit (opc == 10)

LDAPURSH <Xt>, [<Xn|SP>{, #<simm>}]}

```java
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
integer n = \texttt{UInt}(Rn);
integer t = \texttt{UInt}(Rt);
\texttt{AccType} accType = \texttt{AccType\ ORDERED};
\texttt{MemOp} memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then \texttt{MemOp\ LOAD} else \texttt{MemOp\ STORE};
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = \texttt{MemOp\ PREFETCH};
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = \texttt{MemOp\ LOAD};
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 \ll\ scale;
if `HaveMTEExt()` then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[ ];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[ ] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDAPURSW

Load-Acquire RCpc Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

### Assembler Symbols

- `<Xt>`: Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`: Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType accType = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

### Unscaled offset

```
LDAPURSW <Xt>, [<Xn|SP>{, #<simm>}]}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```
if HaveMTEExt() then
    boolean is_load_store = memop IN \{MemOp_STORE, MemOp_LOAD\};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN \{Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP\};
    case c of
        when Constraint_WBSUPPRESS wbback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN \{Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP\};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

### 32-bit (size = 10)

LDAR \(<Wt>\), \([<Xn|SP>|,\#0]\)

### 64-bit (size = 11)

LDAR \(<Xt>\), \([<Xn|SP>|,\#0]\)

---

**Assembler Symbols**

- **<Wt>**
  - Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xt>**
  - Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xn|SP>**
  - Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```asm
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, dbytes, accctype] = data;
  when MemOp_LOAD
    data = Mem[address, dbytes, accctype];
    X[t] = ZeroExtend(data, regsize);
```

---

LDAR
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  |
| size | L  | Rs | o0 | Rt2 |

No offset

LDARB <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
when MemOp_STORE
    data = X[t];
    Mem[address, dbytes, acctype] = data;

when MemOp_LOAD
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

```
LDARH <Wt>, [<Xn|SP>],{,#0}
```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

```
AccType accctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt () then
    SetNotTagCheckedInstruction(n == 31);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
case memop of
    when MemOp STORE
        data = X[t];
        Mem[address, dbytes, accctype] = data;
    when MemOp LOAD
        data = Mem[address, dbytes, accctype];
        X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

32-bit (sz == 0)

LDAXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

LDAXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer datasize = if pair then elsize * 2 else elsize;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDAXP.

Assembler Symbols

<Wt1>  Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>  Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>  Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>  Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExeT() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);
endif

if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  endif
endif

if memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
      when Constraint_NONE rt_unknown = FALSE; // store original value
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
    endwhile
  endif
endif

if s == n && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
    when Constraint_NONE rn_unknown = FALSE; // address is original base
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  endwhile
endif

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];
endif

case memop of
  when MemOp STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    elseif pair then
      bits(datasize DIV 2) el1 = X[t];
      bits(datasize DIV 2) el2 = X[t2];
      data = if BigEndian() then el1 : el2 else el2 : el1;
    else
      data = X[t];
    endif

    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
      // This atomic write will be rejected if it does not refer
      // to the same physical locations after address translation.
      Mem[address, dbytes, acctype] = data;
      status = ExclusiveMonitorsStatus();
      X[s] = ZeroExtend(status, 32);
    endif
  endif
  when MemOp LOAD
    // Tell the Exclusives monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusives monitor will only be set if all the reads are from the
if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        \texttt{X}[t] = \texttt{bits}(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = \texttt{Mem}[address, dbytes, acctype];
        if \texttt{BigEndian}() then
            \texttt{X}[t] = data<datasize-1:elsize>;
            \texttt{X}[t2] = data<elsize-1:0>;
        else
            \texttt{X}[t] = data<elsize-1:0>;
            \texttt{X}[t2] = data<datasize-1:elsize>;
        end
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != \texttt{Align}(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            \texttt{AArch64.Abort}(address, \texttt{AArch64.AlignmentFault}(acctype, iswrite, secondstage));
            \texttt{X}[t] = \texttt{Mem}[address + 0, 8, acctype];
            \texttt{X}[t2] = \texttt{Mem}[address + 8, 8, acctype];
        else
            data = \texttt{Mem}[address, dbytes, acctype];
            \texttt{X}[t] = \texttt{ZeroExtend}(data, regsize);
        end
    end
else
    data = \texttt{Mem}[address, dbytes, acctype];
    \texttt{X}[t] = \texttt{ZeroExtend}(data, regsize);
end

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

32-bit (size == 10)

LDAXR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

LDAXR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
    Constraint c = ConstrainsUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainsUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt Unknown = FALSE; // store original value
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if s == n && n != 31 then
        Constraint c = ConstrainsUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
            when Constraint_NONE rn Unknown = FALSE; // address is original base
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    elsif rn_unknown then
        address = bits(64) UNKNOWN;
    else
        address = X[n];

case memop of
    when MemOp STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elseif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        \( X[t] = \text{bits}(\text{datasize}) \text{ UNKNOWN} \);
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            \( X[t] = \text{data}<\text{datasize}-1:1> \);
            \( X[t2] = \text{data}<1:0> \);
        else
            \( X[t] = \text{data}<1:0> \);
            \( X[t2] = \text{data}<\text{datasize}-1:1> \);
        end if
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            \( X[t] = \text{Mem}[address + 0, 8, \text{acctype}] \);
            \( X[t2] = \text{Mem}[address + 8, 8, \text{acctype}] \);
        else
            data = Mem[address, dbytes, acctype];
            \( X[t] = \text{ZeroExtend}(\text{data}, \text{regsize}) \);
        end if
    end if
end if

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDAXRB**

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

**No offset**

LDAXRB `<Wt>`, [<Xn|SP>{,#0}]

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer datasize = if elsize == 64 then 64 else 32;
```

**Assembler Symbols**

- `<Wt>`    Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);
if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
if memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
      when Constraint_NONE rt_unknown = FALSE; // store original value
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
      when Constraint_NONE rn_unknown = FALSE; // address is original base
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
if n == 31 then
  CheckSPAlignment();
  address = $P[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = $X[n];
case memop of
  when MemOp STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    elseif pair then
      bits(datasize DIV 2) el1 = $X[t];
      bits(datasize DIV 2) el2 = $X[t2];
      data = if BigEndian() then el1 : el2 else el2 : el1;
    else
      data = $X[t];
    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
      // This atomic write will be rejected if it does not refer
      // to the same physical locations after address translation.
      Mem[address, dbytes, acctype] = data;
      status = ExclusiveMonitorsStatus();
      $X[s] = ZeroExtend(status, 32);
    when MemOp LOAD
      // Tell the Exclusives monitors to record a sequence of one or more atomic
      // memory reads from virtual address range [address, address+dbytes-1].
      // The Exclusives monitor will only be set if all the reads are from the
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

No offset

LDAXRH <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer datasize = if pair then elsize * 2 else elsize;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
    Constraint c = ConstrUnpredictable(Unpredictable LDPOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN      rt_unknown = TRUE;  // result is UNKNOWN
        when Constraint UNDEF        UNDEFINED;
        when Constraint NOP          EndOfInstruction();

if memop == MemOp STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrUnpredictable(Unpredictable DATAOVERLAP);
        assert c IN {Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP};
        case c of
            when Constraint UNKNOWN      rt_unknown = TRUE;  // store UNKNOWN value
            when Constraint NONE         rt_unknown = FALSE;  // store original value
            when Constraint UNDEF        UNDEFINED;
            when Constraint NOP          EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrUnpredictable(Unpredictable BASEOVERLAP);
        assert c IN {Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP};
        case c of
            when Constraint UNKNOWN      rn_unknown = TRUE;  // address is UNKNOWN
            when Constraint NONE         rn_unknown = FALSE;  // address is original base
            when Constraint UNDEF        UNDEFINED;
            when Constraint NOP          EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];
        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);
    when MemOp LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
/same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
  if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
  elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, acctype];
    if BigEndian() then
      X[t] = data<datasize-1:elsize>;
      X[t2] = data<elsize-1:0>;
    else
      X[t] = data<elsize-1:0>;
      X[t2] = data<datasize-1:elsize>;
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
      X[t] = Mem[address + 0, 8, acctype];
      X[t2] = Mem[address + 8, 8, acctype];
    else
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtend(data, regsize);
  else // elsize == 64
      // 64-bit load exclusive pair (atomic)
      data = Mem[address, dbytes, acctype];
      X[t] = data<datasize-1:elsize>;
      X[t2] = data<elsize-1:0>;
    else
      X[t] = data<elsize-1:0>;
      X[t2] = data<datasize-1:elsize>;
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
      X[t] = Mem[address + 0, 8, acctype];
      X[t2] = Mem[address + 8, 8, acctype];
    else
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtend(data, regsize);
LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDCLRA and LDCLRAL load from memory with acquire semantics.
- LDCLRL and LDCLRAL store to memory with release semantics.
- LDCLR has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STCLR, STCLRL.

### Integer (ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rn</td>
<td>Rt</td>
<td>Rt</td>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
32-bit LDCLR (size == 10 & A == 0 & R == 0)

LDCLR <Ws>, <Wt>, [<Xn|SP>]

32-bit LDCLRA (size == 10 & A == 1 & R == 0)

LDCLRA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDCLRAL (size == 10 & A == 1 & R == 1)

LDCLRAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDCLRL (size == 10 & A == 0 & R == 1)

LDCLRL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDCLR (size == 11 & A == 0 & R == 0)

LDCLR <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRA (size == 11 & A == 1 & R == 0)

LDCLRA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRAL (size == 11 & A == 1 & R == 1)

LDCLRAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRL (size == 11 & A == 0 & R == 1)

LDCLRL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLR, STCLRL</td>
<td>$A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD   result = data + value;
    when MemAtomicOp_BIC   result = data AND NOT(value);
    when MemAtomicOp_EOR   result = data EOR value;
    when MemAtomicOp_ORR   result = data OR value;
    when MemAtomicOp_SMAX  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN  result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX  result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN  result = if UInt(data) > UInt(value) then value else data;

    Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```
LDCLR, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRLB and LDCLRALB store to memory with release semantics.
- LDCLR has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STCLR, STCLRLB.

```
Integer
(ARMv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
|       0          |       1          |       1          |       0          |       0          |       A          |       R          |       1          |       Rs          |       0          |       0          |       1          |       0          |       Rn           |       Rt           |

size      opc
```

**LDCLR (A == 1 & R == 0)**

```LDCLRAB <Ws>, <Wt>, [<Xn|SP>]```

**LDCLRALB (A == 1 & R == 1)**

```LDCLRALB <Ws>, <Wt>, [<Xn|SP>]```

**LDCLR (A == 0 & R == 0)**

```LDCLR <Ws>, <Wt>, [<Xn|SP>]```

**LDCLRLB (A == 0 & R == 1)**

```LDCLRLB <Ws>, <Wt>, [<Xn|SP>]```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLR, STCLRMB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the following load and store atomically.
// data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDCLRH, LDCLRRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRLH and LDCLRALH store to memory with release semantics.
- LDCLRH has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STCLRH, STCLRHL.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 0  | 1  | 0  | 0  | Rn | 0  | R  | 0  | 0  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 0  | 1  | 0  | 0  | Rn | 0  | R  |

size opc

LDCLRRAH (A == 1 & & R == 0)

LDCLRRAH <Ws>, <Wt>, [<Xn|SP>]

LDCLRALH (A == 1 & & R == 1)

LDCLRALH <Ws>, <Wt>, [<Xn|SP>]

LDCLRH (A == 0 & & R == 0)

LDCLRH <Ws>, <Wt>, [<Xn|SP>]

LDCLRLH (A == 0 & & R == 1)

LDCLRLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

MemAtomicOp op;
case opc of
when '000' op = MemAtomicOp_ADD;
when '001' op = MemAtomicOp_BIC;
when '010' op = MemAtomicOp_EOR;
when '011' op = MemAtomicOp_ORR;
when '100' op = MemAtomicOp_SMAX;
when '101' op = MemAtomicOp_SMIN;
when '110' op = MemAtomicOp_UMAX;
when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLRH, STCLRLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
baby(data) value;
baby(data) data;
baby(data) result;

if HaveMTEExt () then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMN result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDEOR, LDEORA, LDEORAL, LDEORL**

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, **LDEORA** and **LDEORAL** load from memory with acquire semantics.
- **LDEORL** and **LDEORAL** store to memory with release semantics.
- **LDEOR** has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.
For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STEOR, STEORL**.

---

### Integer

**(ARMv8.1)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
</table>

---

LDEOR, LDEORA, LDEORAL, LDEORL
32-bit LDEOR (size == 10 && A == 0 && R == 0)

LDEOR <Ws>, <Wt>, [<Xn|SP>]

32-bit LDEORA (size == 10 && A == 1 && R == 0)

LDEORA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDEORAL (size == 10 && A == 1 && R == 1)

LDEORAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDEORL (size == 10 && A == 0 && R == 1)

LDEORL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDEOR (size == 11 && A == 0 && R == 0)

LDEOR <Xs>, <Xt>, [<Xn|SP>]

64-bit LDEORA (size == 11 && A == 1 && R == 0)

LDEORA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDEORAL (size == 11 && A == 1 && R == 1)

LDEORAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDEORL (size == 11 && A == 0 && R == 1)

LDEORL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;

case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEOR, STEORL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STEORB, STEORLB.

Integer (ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

size | opc

LDEORAB (A == 1 && R == 0)

LDEORAB <Ws>, <Wt>, [<Xn|SP>]

LDEORALB (A == 1 && R == 1)

LDEORALB <Ws>, <Wt>, [<Xn|SP>]

LDEORB (A == 0 && R == 0)

LDEORB <Ws>, <Wt>, [<Xn|SP>]

LDEORB (A == 0 && R == 1)

LDEORB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

MemAtomicOp op;

case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORB, STEORLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDEORH, LDEORAH, LDEORALH, LDEORLH**

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- LDEORH has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STEORH, STEORLH**.

**Integer (ARMv8.1)**

```
           31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
           0  1  1  1  1  0  0  0  A  R  1  |  Rs  |  0  0  1  0  0  0  |  Rn  |  Rt
size      opc
```

**LDEORAH (A == 1 & R == 0)**

LDEORAH <Ws>, <Wt>, [<Xn|SP>]

**LDEORALH (A == 1 & R == 0)**

LDEORALH <Ws>, <Wt>, [<Xn|SP>]

**LDEORH (A == 0 & R == 0)**

LDEORH <Ws>, <Wt>, [<Xn|SP>]

**LDEORLH (A == 0 & R == 1)**

LDEORLH <Ws>, <Wt>, [<Xn|SP>]

```cpp
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORH, STEORLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load Allocation Tag loads an Allocation Tag from a memory address, generates an address with the Logical Address Tag generated from the loaded Allocation Tag, and writes the result to the destination register. The address used for the load is calculated from the source register and an immediate signed offset scaled by the Tag granule.

### Integer

**ARMv8.5**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 1  | imm9 | 0  | 0  | Xn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**LDG**

`<Xt>, [<Xn|SP>{, #<simm>}]`

```plaintext
integer t = Uint(Xt);
integer n = Uint(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
```

### Assembler Symbols

- `<Xt>`: Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- `<simm>`: Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

### Operation

```plaintext
bits(64) address;
bits(4) tag;

if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n];

address = address + offset;
address = Align(address, TAG_GRANULE);
tag = MemTag[address];
address = AddressWithAllocationTag(address, tag);

X[t] = address;
```

---

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDGV

Load Tag Vector loads an IMPLEMENTATION DEFINED number of Allocation Tags from the naturally aligned array of 16 Allocation Tags which includes a tag whose address is the address in the source register, and writes them to the destination register. Bits of the destination register which do not store a tag are set to 0. The Allocation Tag at the address in the source register is always loaded, and the first source register is updated to the address of the first Allocation Tag at an address higher than the original address that was not loaded.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

**Integer**

(ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**Integer**

LDGV <Xt>, [<Xn|SP>]!

integer t = UInt(Xt);
integer n = UInt(Xn);
boolean wback = TRUE;
boolean wb_unknown = FALSE;

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
Operation

bits(64) data = Zeros(64);
bits(64) address;
integer count;

if n == t then
    c = ConstrUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};

    case c of
        when Constraint_WBSUPPRESS
            wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN
            wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF
            UnallocatedEncoding();
        when Constraint_NOP
            EndOfInstruction();

    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    (address, count) = ImpDefArrayStartAndCount(address);

    for i = 0 to count-1
        integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
        bits(4) tag = MemTag[address];
        data<(index*4)+3:(index*4)> = tag;
        address = address + TAG_GRANULE;

    X[t] = data;

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| 1   | x  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | (1)| (1)| (1)| (1)| 0  | (1)| (1)| (1)| (1)| (1)| Rn | |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Rt |

size L Rs o0 Rt2

32-bit (size == 10)

LDLAR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

LDLAR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType.ORDERED;
MemOp memop = if L == '1' then MemOp LOAD else MemOp STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);
endif

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
endif

case memop of
when MemOp_STORE
  data = X[t];
  Mem[address, dbytes, acctype] = data;
when MemOp_LOAD
  data = Mem[address, dbytes, acctype];
  X[t] = ZeroExtend(data, regsize);
endcase

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |

size L Rs o0 Rt2

No offset

LDLARB <\texttt{Wt}>, [<Xn|SP>{{0,#0}}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;

Assembler Symbols

<\texttt{Wt}> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

case memop of
  when MemOp STORE
    data = X[t];
    Mem[address, dbytes, acctype] = data;
  when MemOp LOAD
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

### No offset

**(ARMv8.1)**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)|
LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to
the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional
immediate offset.

For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point Non-temporal pair.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an
attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

\[
\begin{array}{cccccccccccccccccccc}
\text{opc} & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \text{imm7} & \text{Rt2} & \text{Rn} & \text{Rt} \\
\end{array}
\]

32-bit (opc == 00)

LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}] 

64-bit (opc == 01)

LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}] 

128-bit (opc == 10)

LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}] 

boolean wback = FALSE;
boolean postindex = FALSE;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE
behaviors, and particularly LDNP (SIMD&FP).

Assembler Symbols

<Dt1> 
Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt2> 
Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Qt1> 
Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt2> 
Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<St1> 
Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<St2> 
Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Xn|SP> 
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> 
For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0
and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0
and encoded in the "imm7" field as <imm>/8.

For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting
to 0 and encoded in the "imm7" field as <imm>/16.
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VECSTREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '1' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);

Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);
if memop == MemOp_LOAD && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNKNOWN rt_unknown = TRUE;  // result is UNKNOWN
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP EndOfInstruction();
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
if ! postindex then
  address = address + offset;
case memop of
  when MemOp_STORE
data1 = V[t];
data2 = V[t2];
  Mem[address + 0 , dbytes, acctype] = data1;
  Mem[address + dbytes, dbytes, acctype] = data2;
  when MemOp_LOAD
data1 = Mem[address + 0 , dbytes, acctype];
data2 = Mem[address + dbytes, dbytes, acctype];
  if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
    V[t] = data1;
    V[t2] = data2;
  if wback then
    if postindex then
      address = address + offset;
    if n == 31 then
      SP[] = address;
    else
      X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see Load/Store addressing modes. For information about Non-temporal pair instructions, see Load/Store Non-temporal pair.

32-bit (opc == 00)

LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]  

64-bit (opc == 10)

LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]  

boolean wback = FALSE;  
boolean postindex = FALSE;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer t2 = UInt(Rt2);  
AccType acctype = AccType_STREAM;  
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;  
if opc<0> == '1' then UNDEFINED;  
integer scale = 2 + UInt(opc<1>);  
integer datasize = 8 << scale;  
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
bits(64) address;
bite(datasize) data1;
bite(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
    Mem[address + 0 , dbytes, acctype] = data1;
    Mem[address + dbytes, dbytes, acctype] = data2;
    when MemOp_LOAD
    data1 = Mem[address + 0 , dbytes, acctype];
    data2 = Mem[address + dbytes, dbytes, acctype];
    if rt_unknown then
        data1 = bits(datasize) UNKNOWN;
        data2 = bits(datasize) UNKNOWN;
        X[t] = data1;
        X[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset.

### Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| opc | 1   | 0   | 1   | 0   | 0   | 1   | 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

32-bit (opc == 00)

LDP <St1>, <St2>, [<Xn|SP>], #<imm>

64-bit (opc == 01)

LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

128-bit (opc == 10)

LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

```java
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| opc | 1   | 0   | 1   | 0   | 1   | 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

32-bit (opc == 00)

LDP <St1>, <St2>, [<Xn|SP>], #<imm>!

64-bit (opc == 01)

LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>!

128-bit (opc == 10)

LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>!

```java
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| opc | 1   | 0   | 1   | 0   | 0   | 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

LDP (SIMD&FP)
32-bit (opc == 00)

LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}

64-bit (opc == 01)

LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}

128-bit (opc == 10)

LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}

boolean wback = FALSE;
boolean postindex = FALSE;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP (SIMD&FP).

Assembler Symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.
For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.
For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.
For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.
For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bots(64) offset = LSL(SignExtend(imm7, 64), scale);
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = memop IN (MemOp STORE, MemOp LOAD);
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == MemOp LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN (Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP);
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0 , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;
    when MemOp LOAD
        data1 = Mem[address + 0 , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
            V[t] = data1;
            V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

---

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset.

Post-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0 1 0 1 0 0 0 1 1</td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>

64-bit (opc == 10)

LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>!

boolean wback = TRUE;
boolean postindex = TRUE;

Pre-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0 1 0 1 0 0 0 1 1</td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>!

64-bit (opc == 10)

LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>!

boolean wback = TRUE;
boolean postindex = FALSE;

Signed offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0 1 0 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>]

64-bit (opc == 10)

LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>]

boolean wback = FALSE;
boolean postindex = FALSE;
For information about the CONstrained UNPREDICtable behavior of this instruction, see Architectural Constraints on UNPREDICtable behaviors, and particularly LDP.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
binary(64) offset = LSL(SignExtend(imm7, 64), scale);
```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && lwback);

boolean wb_unknown = FALSE;

if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_WB_LOAD);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE;   // value stored is pre-writeback
    when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if ! postindex then
  address = address + offset;

end case memop of
when MemOp_STORE
  if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
  else
    data1 = X[t];
  if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
  else
    data2 = X[t2];
  Mem[address + 0 , dbytes, acctype] = data1;
  Mem[address + dbytes, dbytes, acctype] = data2;
when MemOp_LOAD
  data1 = Mem[address + 0 , dbytes, acctype];
  data2 = Mem[address + dbytes, dbytes, acctype];
  if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
  if signed then
    X[t] = SignExtend(data1, 64);
    X[t2] = SignExtend(data2, 64);
  else
if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| imm7 | Rt2 | Rn | Rt |

```plaintext
opc L
```

### Pre-index

LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

```plaintext
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Signed offset

LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

```plaintext
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDPSW.

### Assembler Symbols

- `<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.
  
  For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if Lopc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
bits(64) address;
bite(s(datasize) data1;
bite(s(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

boolean wb_unknown = FALSE;

if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
  assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint UNDEF UNDEFINED;
    when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
  assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint NONE rt_unknown = FALSE; // value stored is pre-writeback
    when Constraint UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint UNDEF UNDEFINED;
    when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint UNDEF UNDEFINED;
    when Constraint NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = $SP[];
else
  address = $X[n];

if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    if rt_unknown && t == n then
      data1 = bits(datasize) UNKNOWN;
    else
      data1 = $X[t];
    if rt_unknown && t2 == n then
      data2 = bits(datasize) UNKNOWN;
    else
      data2 = $X[t2];
    Mem[address + 0 , dbytes, acctype] = data1;
    Mem[address + dbytes, dbytes, acctype] = data2;
  when MemOp_LOAD
    data1 = Mem[address + 0 , dbytes, acctype];
    data2 = Mem[address + dbytes, dbytes, acctype];
  if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
  if signed then
    $X[t] = SignExtend(data1, 64);
    $X[t2] = SignExtend(data2, 64);
  else
    data1 = Mem[address + 0 , dbytes, acctype];
    data2 = Mem[address + dbytes, dbytes, acctype];
    if rt_unknown then
      data1 = bits(datasize) UNKNOWN;
      data2 = bits(datasize) UNKNOWN;
    if signed then
      $X[t] = SignExtend(data1, 64);
      $X[t2] = SignExtend(data2, 64);
    else
      data1 = Mem[address + 0 , dbytes, acctype];
      data2 = Mem[address + dbytes, dbytes, acctype];
\[ \text{\(X[t]\)} = \text{data1}; \]
\[ \text{\(X[t2]\)} = \text{data2}; \]

if \(\text{wback}\) then
  if \(\text{wb\_unknown}\) then
    \(\text{address} = \text{bits(64)}\) \(\text{UNKNOWN};\)
  elseif \(\text{postindex}\) then
    \(\text{address} = \text{address} + \text{offset};\)
  if \(n = 31\) then
    \(\text{SP[]} = \text{address};\)
  else
    \(\text{X}[n] = \text{address};\)

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

Post-index

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>1</th>
<th>0</th>
<th>imm9</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

8-bit (size == 00 && opc == 01)

LDR <Bt>, [<Xn|SP>], #<simm>

16-bit (size == 01 && opc == 01)

LDR <Ht>, [<Xn|SP>], #<simm>

32-bit (size == 10 && opc == 01)

LDR <St>, [<Xn|SP>], #<simm>

64-bit (size == 11 && opc == 01)

LDR <Dt>, [<Xn|SP>], #<simm>

128-bit (size == 00 && opc == 11)

LDR <Qt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
basic offset = SignExtend(imm9, 64);

Pre-index

| size | 1 | 1 | 1 | 1 | 0 | 0 | x | 1 | 0 | imm9 | 1 | 1 | Rn | Rt |
|------|---|---|---|---|---|---|---|---|------|---|---|----|----|

opc
8-bit (size == 00 && opc == 01)
LDR <Bt>, [<Xn|SP>, #<simm>]

16-bit (size == 01 && opc == 01)
LDR <Ht>, [<Xn|SP>, #<simm>]

32-bit (size == 10 && opc == 01)
LDR <St>, [<Xn|SP>, #<simm>]

64-bit (size == 11 && opc == 01)
LDR <Dt>, [<Xn|SP>, #<simm>]

128-bit (size == 00 && opc == 11)
LDR <Qt>, [<Xn|SP>, #<simm>]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 1 | 1 | 1 | 1 | 0 | 1 | x | 1 | imm12 |
| opc |    | | | | | | | | Rn | | | | | | | | | | | | | | | | | | | | | | | | |
|    | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

8-bit (size == 00 && opc == 01)
LDR <Bt>, [<Xn|SP>{, #<pimm}]}

16-bit (size == 01 && opc == 01)
LDR <Ht>, [<Xn|SP>{, #<pimm}]}

32-bit (size == 10 && opc == 01)
LDR <St>, [<Xn|SP>{, #<pimm}]}

64-bit (size == 11 && opc == 01)
LDR <Dt>, [<Xn|SP>{, #<pimm}]}

128-bit (size == 00 && opc == 11)
LDR <Qt>, [<Xn|SP>{, #<pimm}]}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Sr> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;

LDR (immediate, SIMD&FP)
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

32-bit (size == 10)

LDR <Wt>, [<Xn|SP>], #<simm>

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Pre-index

32-bit (size == 10)

LDR <Wt>, [<Xn|SP>], #<simm>!

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>], #<simm>!

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

32-bit (size == 10)

LDR <Wt>, [<Xn|SP>], #<simm>!

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>], #<simm>!

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
32-bit (size == 10)

LDR <Wt>, [<Xn|SP>{, #<pimm>}]}

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>{, #<pimm>}]}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDR (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN (MemOp STORE, MemOp LOAD);
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bite(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN (Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP);
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN (Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP);
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>opc</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

LDR <St>, <label>

64-bit (opc == 01)

LDR <Dt>, <label>

128-bit (opc == 10)

LDR <Qt>, <label>

integer t = `UInt`(Rt);
integer size;
bits(64) offset;
case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 16;
  when '11'
    UNDEFINED;
offset = `SignExtend`(imm19:'00', 64);

Assembler Symbols

<Dt> Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

bits(64) address = PC[] + offset;
bits(size*8) data;

`CheckFPAdvSIMDEnabled64`();
data = Mem[address, size, AccType_VEC];
V[t] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>imm19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

opc

32-bit (opc == 00)

LDR <Wt>, <label>

64-bit (opc == 01)

LDR <Xt>, <label>

```java
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xt>`: Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<label>`: Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```java
bits(64) address = PC[] + offset;
bits(size*8) data;

case memop of
  when MemOp_LOAD
    data = Mem[address, size, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, 64);
    else
      X[t] = data;
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);
```

LDR (literal)  Page 711
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| size | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| opc  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

8-fsreg,LDR-8-fsreg (size == 00 && opc == 01 && option != 011)

LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}] 

8-fsreg,LDR-8-fsreg (size == 00 && opc == 01 && option == 011)

LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]

16-fsreg,LDR-16-fsreg (size == 01 && opc == 01)

LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

32-fsreg,LDR-32-fsreg (size == 10 && opc == 01)

LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

64-fsreg,LDR-64-fsreg (size == 11 && opc == 01)

LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

128-fsreg,LDR-128-fsreg (size == 00 && opc == 11)

LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
Integer shift = if S == '1' then scale else 0;

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rt" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> For the 8-bit variant: is the index extend specifier, encoded in "option":

option <extend>
010  UXTW
110  SXTW
111  SXTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option <extend>
010  UXTW
011  LSL
110  SXTW
111  SXTX

<amount>

For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

\[
\begin{array}{c|c}
S & <amount> \\
\hline
0 & \#0 \\
1 & \#1 \\
\end{array}
\]

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

\[
\begin{array}{c|c}
S & <amount> \\
\hline
0 & \#0 \\
1 & \#2 \\
\end{array}
\]

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

\[
\begin{array}{c|c}
S & <amount> \\
\hline
0 & \#0 \\
1 & \#3 \\
\end{array}
\]

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

\[
\begin{array}{c|c}
S & <amount> \\
\hline
0 & \#0 \\
1 & \#4 \\
\end{array}
\]

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
Integer datasize = 8 << scale;
```
Operation

```cpp
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOpSTORE, MemOpLOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if ! postindex then
    address = address + offset;
case memop of
    when MemOpSTORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;
    when MemOpLOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Rm</td>
<td>option</td>
</tr>
<tr>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>

32-bit (size == 10)

LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;               // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

Assembler Symbols

<Wt> is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
  when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
  when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP EndOfInstruction();

if memop == MemOp STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
  when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
  when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP EndOfInstruction();

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if ! postindex then
  address = address + offset;

  case memop of
  when MemOp STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
      Mem[address, datasize DIV 8, acctype] = data;

  when MemOp LOAD
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRAA, LDRAB

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the
specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a
register.

Key A is used for LDRAA, and key B is used for LDRAB.

If the authentication passes, the PE behaves the same as for an LDR instruction. If the authentication fails, a Translation fault is generated.
The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the
address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see Load/Store addressing modes.

Unscaled offset
(ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 0  | M  | S  | 1  | imm9 | W  | 1  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size

Key A, offset (M == 0 & W == 0)

LDRAA <Xt>, [<Xn|SP>{, #<simm>}]

Key A, pre-indexed (M == 0 & W == 1)

LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!

Key B, offset (M == 1 & W == 0)

LDRAB <Xt>, [<Xn|SP>{, #<simm>}]

Key B, pre-indexed (M == 1 & W == 1)

LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!

if !HavePACExt() || size != '11' then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
integer scale = 3;
bits(64) offset = LSL(SignExtend(S10, 64), scale);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the
"S:imm9" field as <simm>/8.
Operation

```c
bits(64) address;
bias(64) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if use_key_a then
    address = AuthDA(address, X[31]);
else
    address = AuthDB(address, X[31]);

address = address + offset;
data = Mem[address, 8, AccType_NORMAL];
X[t] = data;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDRB (immediate)**

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

### Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 1   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | imm9 | 0   | 1   | Rn  | Rt  |

**size**  **opc**  **Rt**

### Pre-index

```java
LDRB <Wt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 1   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | imm12 | | 1   | 1   | Rn  | Rt  |

**size**  **opc**  **Rt**

### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRH (immediate)*.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

**Shared Decode**

```plaintext
class AccType { normal, extended; }
enum MemOp { LOAD, STORE; }

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType.NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;  // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

    case memop of
        when MemOp STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;

        when MemOp LOAD
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);
        if wback then
            if wb_unknown then
                address = bits(64) UNKNOWN;
            elseif postindex then
                address = address + offset;
            if n == 31 then
                SP[] = address;
            else
                X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

Extended register (option != 011)

\[
\text{LDRB } <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}] 
\]

Shifted register (option == 011)

\[
\text{LDRB } <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}] 
\]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;             // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> Is the index extend specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
Operation

```c
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    end case;

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    end case;

if n == 31 then
    if memop != MemOp_PREFETCH then
        address = SP[];
    else
        address = X[n];
    end
    if ! postindex then
        address = address + offset;
    end
end case;
memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
        end
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
        end
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
if wback then
    if wbUnknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    end
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
    end
end
```

LDRB (register)
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size | opc | imm9 | 0 | 1 | Rn | Rt

Pre-index

LDRH <Wt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size | opc | imm9 | 1 | 1 | Rn | Rt

Pre-index

LDRH <Wt>, [<Xn|SP>], #<simm>!

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size | opc | imm12 | Rn | Rt

Unsigned offset

LDRH <Wt>, [<Xn|SP>{, #<pimm>}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSH(ZeroExtend(imm12, 64), scale);

Assembler Symbols

<Wt>   Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

**Shared Decode**

```c
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

32-bit

\[
\text{LDRH} \ <Wt>, \ [<Xn|SP>], \ (<Wm>|<Xm>){, \ <extend> \ {<amount>}}
\]

```plaintext
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>`: When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>`: When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>`: Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when `<amount>` is omitted. It is encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- `<amount>`: Is the index shift amount, optional only when `<extend>` is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType.NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];
    endif

    if ! postindex then
        address = address + offset;
    endif

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
        endif

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
        endif

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
        endif
    endif
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | x  | 0  | 0  | 0  | 1  | 1  | imm9| 0  | 1  | Rn | Rn | Rt | Rt |

32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<simm>

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<simm>

```java
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | x  | 0  | 0  | 0  | 1  | 1  | imm9| 1  | 1  | Rn | Rn | Rt | Rt |

32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<simm>!

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<simm>!

```java
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | x  | 0  | 0  | 1  | imm12| Rn | Rn | Rt | Rt |

LDRSB (immediate)
32-bit (opc == 11)
LDRSB <Wt>, [<Xn|SP>!, #<pimm>]

64-bit (opc == 10)
LDRSB <Xt>, [<Xn|SP>!, #<pimm>]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSB (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
if \texttt{HaveMTEExt()} then
    boolean \texttt{is\_load\_store} = \textop{memop} \in \{\texttt{MemOp\_STORE}, \texttt{MemOp\_LOAD}\};
    \texttt{SetNotTagCheckedInstruction} (\texttt{is\_load\_store} && \texttt{n} == 31 && !\texttt{wback});

bits(64) \texttt{address};
bits(\texttt{datasize}) \texttt{data};

boolean \texttt{wb\_unknown} = FALSE;
boolean \texttt{rt\_unknown} = FALSE;

if \texttt{memop} == \texttt{MemOp\_LOAD} && \texttt{wback} && \texttt{n} == \texttt{t} && \texttt{n} != 31 then
    \texttt{c} = \texttt{ConstrainUnpredictable} (\texttt{Unpredictable\_WBOVERLAPL});
    assert \texttt{c} \in \{\texttt{Constraint\_WBSUPPRESS}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDEF}, \texttt{Constraint\_NOP}\};
    case \texttt{c} of
        \texttt{Constraint\_WBSUPPRESS} \texttt{wback} = FALSE; \quad // writeback is suppressed
        \texttt{Constraint\_UNKNOWN} \texttt{wb\_unknown} = TRUE; \quad // writeback is UNKNOWN
        \texttt{Constraint\_UNDEF} \texttt{UNDEFINED};
        \texttt{Constraint\_NOP} \texttt{EndOfInstruction}();

if \texttt{memop} == \texttt{MemOp\_STORE} && \texttt{wback} && \texttt{n} == \texttt{t} && \texttt{n} != 31 then
    \texttt{c} = \texttt{ConstrainUnpredictable} (\texttt{Unpredictable\_WBOVERLAPST});
    assert \texttt{c} \in \{\texttt{Constraint\_NONE}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDEF}, \texttt{Constraint\_NOP}\};
    case \texttt{c} of
        \texttt{Constraint\_NONE} \texttt{rt\_unknown} = FALSE; \quad // value stored is original value
        \texttt{Constraint\_UNKNOWN} \texttt{rt\_unknown} = TRUE; \quad // value stored is UNKNOWN
        \texttt{Constraint\_UNDEF} \texttt{UNDEFINED};
        \texttt{Constraint\_NOP} \texttt{EndOfInstruction}();

if \texttt{n} == 31 then
    if \texttt{memop} != \texttt{MemOp\_PREFETCH} then \texttt{CheckSPAlignment}();
    \texttt{address} = \texttt{SP}[];
else
    \texttt{address} = \texttt{X}[\texttt{n}];

if ! \texttt{postindex} then
    \texttt{address} = \texttt{address} + \texttt{offset};

case \texttt{memop} of
    \texttt{MemOp\_STORE}
        if \texttt{rt\_unknown} then
            \texttt{data} = \texttt{bits(\texttt{datasize}) UNKNOWN};
        else
            \texttt{data} = \texttt{X}[\texttt{t}];
            \texttt{Mem}[\texttt{address}, \texttt{datasize} \texttt{DIV} 8, \texttt{acctype}] = \texttt{data};
    \texttt{MemOp\_LOAD}
        \texttt{data} = \texttt{Mem}[\texttt{address}, \texttt{datasize} \texttt{DIV} 8, \texttt{acctype}];
        if \texttt{signed} then
            \texttt{X}[\texttt{t}] = \texttt{SignExtend} (\texttt{data}, \texttt{regsize});
        else
            \texttt{X}[\texttt{t}] = \texttt{ZeroExtend} (\texttt{data}, \texttt{regsize});
    \texttt{MemOp\_PREFETCH}
        \texttt{Prefetch} (\texttt{address}, \texttt{t<4:0>});

if \texttt{wback} then
    if \texttt{wb\_unknown} then
        \texttt{address} = \texttt{bits(64)} UNKNOWN;
    elseif \texttt{postindex} then
        \texttt{address} = \texttt{address} + \texttt{offset};
    if \texttt{n} == 31 then
        \texttt{SP}[\texttt{]} = \texttt{address};
    else
        \texttt{X}[\texttt{n}] = \texttt{address};
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
## LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see \textit{Load/Store addressing modes}.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 1  | 1  | 0  | 0  | 1  |   x | 1 |   Rm | option | S  | 1  | 0  |
| size| opc |
```

### 32-bit with extended register offset (opc == 11 && option != 011)

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### 32-bit with shifted register offset (opc == 11 && option == 011)

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### 64-bit with extended register offset (opc == 10 && option != 011)

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### 64-bit with shifted register offset (opc == 10 && option == 011)

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```python
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;             // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

### Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>`: Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>`: When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>`: When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>`: Is the index extend specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- `<amount>`: Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
Operation

bits(64) offset = \texttt{ExtendReg}(m, extend_type, shift);
if \texttt{HaveMTEExt}() then
  boolean is_load_store = memop IN \{\texttt{MemOp\_STORE}, \texttt{MemOp\_LOAD}\};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
if memop == \texttt{MemOp\_LOAD} && wback && n == t && n != 31 then
  c = \texttt{ConstrainUnpredictable}(\texttt{Unpredictable\_WBOVERLAPLD});
  assert c IN \{\texttt{Constraint\_WBSUPPRESS}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDEF}, \texttt{Constraint\_NOP}\};
  case c of
    when \texttt{Constraint\_WBSUPPRESS} wback = FALSE; // writeback is suppressed
    when \texttt{Constraint\_UNKNOWN} wbUnknown = TRUE; // writeback is UNKNOWN
    when \texttt{Constraint\_UNDEF} UNDEFINED;
    when \texttt{Constraint\_NOP} EndOfInstruction();
  end;
if memop == \texttt{MemOp\_STORE} && wback && n == t && n != 31 then
  c = \texttt{ConstrainUnpredictable}(\texttt{Unpredictable\_WBOVERLAPST});
  assert c IN \{\texttt{Constraint\_NONE}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDEF}, \texttt{Constraint\_NOP}\};
  case c of
    when \texttt{Constraint\_NONE} rtUnknown = FALSE;  // value stored is original value
    when \texttt{Constraint\_UNKNOWN} rtUnknown = TRUE;  // value stored is UNKNOWN
    when \texttt{Constraint\_UNDEF} UNDEFINED;
    when \texttt{Constraint\_NOP} EndOfInstruction();
  end;
if n == 31 then
  if memop != \texttt{MemOp\__PREFETCH} then \texttt{CheckSPAlignment}();
  address = SP[];
else
  address = X[n];
if ! postindex then
  address = address + offset;
case memop of
  when \texttt{MemOp\_STORE}
    if rtUnknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
      Mem\[address, datasize DIV 8, acctype\] = data;
  when \texttt{MemOp\_LOAD}
    data = Mem\[address, datasize DIV 8, acctype\];
    if signed then
      X[t] = \texttt{SignExtend}(data, regsize);
    else
      X[t] = \texttt{ZeroExtend}(data, regsize);
  when \texttt{MemOp\__PREFETCH}
    Prefetch(address, t<4:0>);
if wback then
  if wbUnknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | x  | 0  | 0  | 1  | x  | 0  | 0  | 1  | x  | 0  | 0  | 1  | x  | 0  | 0  | 1  | x  | 0  | 0  | 1  | Rn |
| size | opc |

32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>], #<simm>

64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bias(64) offset = SignExtend(imm9, 64);

Pre-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | x  | 0  | 0  | 1  | x  | 0  | 0  | 1  | x  | 0  | 0  | 1  | x  | 0  | 0  | 1  | x  | 0  | 0  | 1  | Rn |
| size | opc |

32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>], #<simm>!

64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>], #<simm>!

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bias(64) offset = SignExtend(imm9, 64);

Unsigned offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | x  | 0  | 0  | 0  | 1  | x  | 0  | 0  | 0  | 1  | x  | 0  | 0  | 0  | 1  | x  | 0  | 0  | 0  | 1  | imm12 |
| size | opc |

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bias(64) offset = SignExtend(imm9, 64);
32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>{, #<pimm>}

64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>{, #<pimm>}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSH (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

integer n = Uint(Rn);
integer t = Uint(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bv(data) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(data) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see Load/Store addressing modes.

32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> []<amount>}]}

64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> []<amount>}]}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;             // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

.getAmount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>
integer n = \texttt{UInt}(Rn);
integer t = \texttt{UInt}(Rt);
integer m = \texttt{UInt}(Rm);
AccType acctype = \texttt{AccType\_NORMAL};
MemOp memop;
boolean signed;
integer regsize;
if \texttt{opc<1>} == '0' then
  // store or zero-extending load
  memop = if \texttt{opc<0>} == '1' then \texttt{MemOp\_LOAD} else \texttt{MemOp\_STORE};
  regsize = if \texttt{size} == '11' then 64 else 32;
  signed = FALSE;
else
  if \texttt{size} == '11' then
    memop = \texttt{MemOp\_PREFETCH};
    if \texttt{opc<0>} == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = \texttt{MemOp\_LOAD};
    if \texttt{size} == '10' \&\& \texttt{opc<0>} == '1' then UNDEFINED;
    regsize = if \texttt{opc<0>} == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 \ll \texttt{scale};
Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    bits(64) address;
    bits(datasize) data;
    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE;  // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[);
    else
        address = X[n];

    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);
    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```

LDRSH (register)
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | imm9 |    | 0  | 1  |    |    |    |    |    |    |    |    |    |    |
| size |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opc  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

```
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

`boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | imm9 | 1  | 1  |    |    |    |    |    |    |    |    |    |
| size |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opc  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

```
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

`boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | imm12 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opc  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}
```

`boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSW (immediate).

### Assembler Symbols

- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then UNDEFINED;
    else // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```
if \texttt{HaveMTETExt()} then
\begin{verbatim}
  boolean is_load_store = memop IN \{\texttt{MemOp STORE, MemOp LOAD}\};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

  bits(64) address;
  bits(datasize) data;

  boolean wb_unknown = FALSE;
  boolean rt_unknown = FALSE;

  if memop == \texttt{MemOp LOAD} && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN \{\texttt{Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP}\};
    case c of
      when \texttt{Constraint_WBSUPPRESS} wback = FALSE; // writeback is suppressed
      when \texttt{Constraint_UNKNOWN} wb_unknown = TRUE; // writeback is UNKNOWN
      when \texttt{Constraint_UNDEF} UNDEFINED;
      when \texttt{Constraint_NOP} EndOfInstruction();
  \end{verbatim}

\begin{verbatim}
  if memop == \texttt{MemOp STORE} && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN \{\texttt{Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP}\};
    case c of
      when \texttt{Constraint_NONE} rt_unknown = FALSE;  // value stored is original value
      when \texttt{Constraint_UNKNOWN} rt_unknown = TRUE; // value stored is UNKNOWN
      when \texttt{Constraint_UNDEF} UNDEFINED;
      when \texttt{Constraint_NOP} EndOfInstruction();
  \end{verbatim}

\begin{verbatim}
  if n == 31 then
    if memop != \texttt{MemOp_PREFETCH} then CheckSPAlignment();
    address = \texttt{SP}[];
  else
    address = \texttt{X}[n];

  if ! postindex then
    address = address + offset;

  case memop of
    when \texttt{MemOp_STORE}
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = \texttt{X}[t];
        \texttt{Mem}[address, datasize DIV 8, acctype] = data;
    when \texttt{MemOp_LOAD}
      data = \texttt{Mem}[address, datasize DIV 8, acctype];
      if signed then
        \texttt{X}[t] = \texttt{SignExtend}(data, regsize);
      else
        \texttt{X}[t] = \texttt{ZeroExtend}(data, regsize);
    when \texttt{MemOp_PREFETCH}
      \texttt{Prefetch}(address, t<4:0>);
  \end{verbatim}

\begin{verbatim}
  if wback then
    if wb Unknown then
      address = bits(64) UNKNOWN;
    elsif postindex then
      address = address + offset;
    if n == 31 then
      \texttt{SP}[] = address;
    else
      \texttt{X}[n] = address;
\end{verbatim}
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

Literal

\[
\text{LDRSW } <\text{Xt}>, <\text{label}>
\]

\[
\begin{align*}
\text{integer } t &= \text{UInt}(Rt); \\
\text{MemOp } memop &= \text{MemOp\ LOAD}; \\
\text{boolean } signed &= \text{FALSE}; \\
\text{integer } size; \\
\text{bits}(64) \text{ offset}; \\
\text{case } opc \text{ of} \\
& \quad \text{when '00'} \\
& \quad \quad \text{size} = 4; \\
& \quad \text{when '01'} \\
& \quad \quad \text{size} = 8; \\
& \quad \text{when '10'} \\
& \quad \quad \text{size} = 4; \\
& \quad \quad \text{signed} = \text{TRUE}; \\
& \quad \text{when '11'} \\
& \quad \quad \text{memop} = \text{MemOp\ PREFETCH}; \\
& \quad \text{offset} = \text{SignExtend}(\text{imm19:'00'}, 64);
\end{align*}
\]

Assembler Symbols

\(<\text{Xt}>\) Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

\(<\text{label}>\) Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

\[
\begin{align*}
\text{bits}(64) \text{ address} &= \text{PC[]} + \text{offset}; \\
\text{bits(size*8) data}; \\
\text{case memop of} \\
& \quad \text{when MemOp\ LOAD} \\
& \quad \quad \text{data} = \text{Mem}[\text{address, size, AccType\ NORMAL}]; \\
& \quad \quad \text{if signed then} \\
& \quad \quad \quad \text{X}[t] = \text{SignExtend}(\text{data}, 64); \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{X}[t] = \text{data}; \\
& \quad \text{when MemOp\ PREFETCH} \\
& \quad \quad \text{Prefetch}(\text{address, t<4:0>});
\end{align*}
\]

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | Rm | option | S | 1 | 0 | Rn | | Rt |

size opc

64-bit

LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>
Shared Decode

```c
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if ! postindex then
    address = address + offset;
    case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSETA and LDSETAL load from memory with acquire semantics.
- LDSETL and LDSETAL store to memory with release semantics.
- LDSET has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSET, STSETL.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size opc
32-bit LDSET (size == 10 && A == 0 && R == 0)
   LDSET <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSETA (size == 10 && A == 1 && R == 0)
   LDSETA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSETAL (size == 10 && A == 1 && R == 1)
   LDSETAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSETL (size == 10 && A == 0 && R == 1)
   LDSETL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDSET (size == 11 && A == 0 && R == 0)
   LDSET <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETA (size == 11 && A == 1 && R == 0)
   LDSETA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETAL (size == 11 && A == 1 && R == 1)
   LDSETAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETL (size == 11 && A == 0 && R == 1)
   LDSETL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSET, STSETL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

#### Operation

- `bits(64)` address;
- `bits(datasize)` value;
- `bits(datasize)` data;
- `bits(datasize)` result;

if `HaveMTEExt()` then
  `SetNotTagCheckedInstruction(n == 31);`

value = $X[s]$;
if $n == 31$ then
  `CheckSPAlignment();`
  address = $SP[]$;
else
  address = $X[n]$;

// All observers in the shareability domain observe the
// following load and store atomically.
data = `Mem[address, datasize DIV 8, ldacctype];`

case op of
  when MemAtomicOp_ADD result = data + value;
  when MemAtomicOp_BIC result = data AND NOT(value);
  when MemAtomicOp_EOR result = data EOR value;
  when MemAtomicOp_ORR result = data OR value;
  when MemAtomicOp_SMAX result = if $SInt$(data) > $SInt$(value) then data else value;
  when MemAtomicOp_SMIN result = if $SInt$(data) > $SInt$(value) then value else data;
  when MemAtomicOp_UMAX result = if $UInt$(data) > $UInt$(value) then data else value;
  when MemAtomicOp_UMIN result = if $UInt$(data) > $UInt$(value) then value else data;

  Mem[address, datasize DIV 8, stacctype] = result;

if $t != 31$ then
  $X[t] = ZeroExtend(data, regsize);$

#### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LDSETB, LDSETAB, LDSETALB, LDSETLB**

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, **LDSETAB** and **LDSETALB** load from memory with acquire semantics.
- **LDSETLB** and **LDSETALB** store to memory with release semantics.
- **LDSETB** has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STSETB, STSETLB**.

### Integer

**(ARMv8.1)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

size opc

**LDSETAB (A == 1 && R == 0)**

LDSETAB <Ws>, <Wt>, [<Xn|SP>]

**LDSETALB (A == 1 && R == 1)**

LDSETALB <Ws>, <Wt>, [<Xn|SP>]

**LDSETB (A == 0 && R == 0)**

LDSETB <Ws>, <Wt>, [<Xn|SP>]

**LDSETLB (A == 0 && R == 1)**

LDSETLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

t = UInt(Rt);
n = UInt(Rn);
s = UInt(Rs);

datasize = 8 << UInt(size);
regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;

case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETB, STSETLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt () then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD        result = data + value;
    when MemAtomicOp_BIC        result = data AND NOT(value);
    when MemAtomicOp_EOR        result = data EOR value;
    when MemAtomicOp_ORR        result = data OR value;
    when MemAtomicOp_SMAX       result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN       result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX       result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN       result = if UInt(data) > UInt(value) then value else data;

    Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSETH, LDSETAH, LDSETELH, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, \texttt{LDSETAH} and \texttt{LDSETELH} load from memory with acquire semantics.
- \texttt{LDSETLH} and \texttt{LDSETELH} store to memory with release semantics.
- \texttt{LDSETH} has no memory ordering requirements.

For more information about memory ordering semantics see \textit{Load-Acquire, Store-Release}. For information about memory accesses see \textit{Load/Store addressing modes}.

This instruction is used by the alias \texttt{STSETH, STSETLH}.

### Integer (ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| \texttt{A}     | \texttt{R}    | \texttt{1}     | \texttt{Rs}    | \texttt{0}     | \texttt{0}     | \texttt{Wt}    | \texttt{0}     | \texttt{0}     | \texttt{1}     | \texttt{1}     | \texttt{n}     | \texttt{Xn}    | \texttt{SP}    | \texttt{0}     | \texttt{0}     | \texttt{1}     | \texttt{1}     | \texttt{0}     | \texttt{0}     | \texttt{Rn}    | \texttt{Rt}    |

\texttt{LDSETH} \texttt{(A == 0 && R == 0)}

\texttt{LDSETH} <\texttt{Ws}>, <\texttt{Wt}>, [<\texttt{Xn}\mid\texttt{SP}>]

\texttt{LDSETH} \texttt{(A == 0 && R == 1)}

\texttt{LDSETH} <\texttt{Ws}>, <\texttt{Wt}>, [<\texttt{Xn}\mid\texttt{SP}>]

\texttt{LDSETH} \texttt{(A == 0 && R == 0)}

\texttt{LDSETH} <\texttt{Ws}>, <\texttt{Wt}>, [<\texttt{Xn}\mid\texttt{SP}>]

\texttt{LDSETLH} \texttt{(A == 0 && R == 1)}

\texttt{LDSETLH} <\texttt{Ws}>, <\texttt{Wt}>, [<\texttt{Xn}\mid\texttt{SP}>]

\begin{verbatim}
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
\end{verbatim}

\section*{Assembler Symbols}

<\texttt{Ws}> 
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETH, STSETLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
b instability(datasize) value;
b instability(datasize) data;
b instability(datasize) result;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
  CheckSPAimgement();
  address = SP[];
else
  address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
  when MemAtomicOp_ADD result = data + value;
  when MemAtomicOp_BIC result = data AND NOT(value);
  when MemAtomicOp_EOR result = data EOR value;
  when MemAtomicOp_ORR result = data OR value;
  when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
  X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMAXA and LDSMAXAL load from memory with acquire semantics.
- LDSMAXL and LDSMAXAL store to memory with release semantics.
- LDSMAX has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMAX, STSMAXL.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | x  | 1  | R  | 1  | Rs | 0  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | Rn |
| size |         |           |     | opc |

LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL
32-bit LDSMAX (size == 10 && A == 0 && R == 0)

LDSMAX <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMAXA (size == 10 && A == 1 && R == 0)

LDSMAXA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMAXAL (size == 10 && A == 1 && R == 1)

LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMAXL (size == 10 && A == 0 && R == 1)

LDSMAXL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDSMAX (size == 11 && A == 0 && R == 0)

LDSMAX <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMAXA (size == 11 && A == 1 && R == 0)

LDSMAXA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMAXAL (size == 11 && A == 1 && R == 1)

LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMAXL (size == 11 && A == 0 && R == 1)

LDSMAXL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = Uint(Rt);
integer n = Uint(Rn);
integer s = Uint(Rs);

integer datasize = 8 << Uint(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
when '000' op = MemAtomicOp_ADD;
when '001' op = MemAtomicOp_BIC;
when '010' op = MemAtomicOp_EOR;
when '011' op = MemAtomicOp_ORR;
when '100' op = MemAtomicOp_SMAX;
when '101' op = MemAtomicOp_SMIN;
when '110' op = MemAtomicOp_UMAX;
when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rs" field.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAX, STSMAXL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = \text{X}[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = \text{X}[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = \text{Mem}[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

\text{Mem}[address, datasize DIV 8, stacctype] = result;
if t != 31 then
    \text{X}[t] = \text{ZeroExtend}(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

---

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00beta8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMAXB, STSMAXLB.

**Integer (ARMv8.1)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 0  | 0  | Rn | R  |
| size | opc |

**LDSMAXB (A == 1 && R == 0)**

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXALB (A == 1 && R == 1)**

LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXB (A == 0 && R == 0)**

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXLB (A == 0 && R == 1)**

LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]

```plaintext
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datatype = 8 << UInt(size);
integer regsize = if datatype == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAxB, STSMAxLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAHeight();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMAXH, STSMAXLH.

### Integer

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | | Rs  | 0  | 1  | 0  | 0  | 0  | Rn  | |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| size | opc |

**LDSMAXAH (A == 1 && R == 0)**

LDSMAXAH \(<Ws>, <Wt>, [<Xn|SP>]\)

**LDSMAXALH (A == 1 && R == 1)**

LDSMAXALH \(<Ws>, <Wt>, [<Xn|SP>]\)

**LDSMAXH (A == 0 && R == 0)**

LDSMAXH \(<Ws>, <Wt>, [<Xn|SP>]\)

**LDSMAXLH (A == 0 && R == 1)**

LDSMAXLH \(<Ws>, <Wt>, [<Xn|SP>]\)

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

### Assembler Symbols

\(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAXH, STSMAXLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

    Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, **LDSMINA** and **LDSMINAL** load from memory with acquire semantics.
- **LDSMINL** and **LDSMINAL** store to memory with release semantics.
- **LDSMIN** has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.
For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STSMIN, STSMINL**.

### Integer

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

| size | opc |
32-bit LDSMIN (size == 10 && A == 0 && R == 0)
  LDSMIN <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMINA (size == 10 && A == 1 && R == 0)
  LDSMINA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMINAL (size == 10 && A == 1 && R == 1)
  LDSMINAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMINL (size == 10 && A == 0 && R == 1)
  LDSMINL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDSMIN (size == 11 && A == 0 && R == 0)
  LDSMIN <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINA (size == 11 && A == 1 && R == 0)
  LDSMINA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINAL (size == 11 && A == 1 && R == 1)
  LDSMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINL (size == 11 && A == 0 && R == 1)
  LDSMINL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
  case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt>  Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs>  Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt>  Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMIN, STSMINL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

    Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

---

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB**

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMINB, STSMINLB.

**Integer (ARMv8.1)**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

- **size**
- **opc**

**LDSMINAB (A == 1 && R == 0)**

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

**LDSMINALB (A == 1 && R == 1)**

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

**LDSMINB (A == 0 && R == 0)**

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

**LDSMINLB (A == 0 && R == 1)**

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

if !HaveAtomicExt() then UNDEFINED;

t = UInt(Rt);
n = UInt(Rn);
s = UInt(Rs);

datasize = 8 << UInt(size);
regsize = if datasize == 64 then 64 else 32;

```
MemAtomicOp op;

case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

**Assembler Symbols**

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMINB, STSMINLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

    Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMINH, STSMINLH.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn | 0  | Re |

size opc

LDSMINAH (A == 1 && R == 0)

LDSMINAH <Ws>, <Wt>, [<Xn|SP>]

LDSMINALH (A == 1 && R == 1)

LDSMINALH <Ws>, <Wt>, [<Xn|SP>]

LDSMINH (A == 0 && R == 0)

LDSMINH <Ws>, <Wt>, [<Xn|SP>]

LDSMINLH (A == 0 && R == 1)

LDSMINLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMINH, STSMINLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if Uint(data) > Uint(value) then data else value;
    when MemAtomicOp_UMIN result = if Uint(data) > Uint(value) then value else data;

    Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

---

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
• The instruction is executed at EL1.
• The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x</td>
<td>0 0</td>
</tr>
<tr>
<td>imm9</td>
<td>1 0</td>
</tr>
<tr>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>

32-bit (size == 10)

LDTR <Wt>, [<Xn|SP>{, #<simm>}]

64-bit (size == 11)

LDTR <Xt>, [<Xn|SP>{, #<simm>}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
unpriv_at_el2 = HaveEL(EL2) && HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    bits(64) address;
    bits(datasize) data;

    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if memop == MemOp LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
        when MemOp LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = HaveEL(EL2) && HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainingUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainingUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[);
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[) = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
• The instruction is executed at EL1.
• The instruction is executed at EL2 when the Effective value of HCR_EL2.\{E2H, TGE\} is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see "Load/Store addressing modes."

Unscaled offset

LDTRH <Wt>, [<Xn|SP>\{, #<simm>\}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = HaveEL(EL2) && HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if (user_access_override && (unpriv_at_el1 || unpriv_at_el2)) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

    integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop in {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    bits(64) address;
    bits(datasize) data;

    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c in {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c in {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[ ];
    else
        address = X[n];

    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[ ] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDTRSB**

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>x</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>imm9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Rn</td>
<td>Rn</td>
</tr>
<tr>
<td>Rt</td>
<td>Rt</td>
</tr>
</tbody>
</table>

### 32-bit (opc == 11)

LDTRSB `<Wt>`, `[<Xn|SP>{, #<simm}>]

### 64-bit (opc == 10)

LDTRSB `<Xt>`, `[<Xn|SP>{, #<simm}>]

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && ! (EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '1');
unpriv_at_el2 = HaveEL(EL2) && HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '1';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
if haveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_PREFETCH then
    address = SP[ ];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

### Assembler Symbols

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

```plaintext
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '1' &&
unpriv_at_el2 = HaveEL(EL2) && HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '1' &&
user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wbback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

Assembler Symbols

- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = HaveEL(EL2) && HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then UNDEFINED;
else
  // sign-extending load
  memop = MemOp_LOAD;
  if size == '10' && opc<0> == '1' then UNDEFINED;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

integer datasize = 8 << scale;
```

Unscaled offset

LDTRSW `<Xt>`, `[<Xn|SP>{, #<simm>}]`

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
if `HaveMTEExt()` then
    boolean is_load_store = memop IN `{MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    bits(64) address;
    bits(datasize) data;

    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if memop == MemOp_LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN `{Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
    
    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN `{Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
    
    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
        
        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);
    
    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMAXA and LDUMAXAL load from memory with acquire semantics.
- LDUMAXL and LDUMAXAL store to memory with release semantics.
- LDUMAX has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMAX, STUMAXL.

### Integer (ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 0  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL
32-bit LDUMAX (size == 10 & A == 0 & R == 0)

LDUMAX <Ws>, <Wt>, [Xn|SP]

32-bit LDUMAXA (size == 10 & A == 1 & R == 0)

LDUMAXA <Ws>, <Wt>, [Xn|SP]

32-bit LDUMAXAL (size == 10 & A == 1 & R == 1)

LDUMAXAL <Ws>, <Wt>, [Xn|SP]

32-bit LDUMAXL (size == 10 & A == 0 & R == 1)

LDUMAXL <Ws>, <Wt>, [Xn|SP]

64-bit LDUMAX (size == 11 & A == 0 & R == 0)

LDUMAX <Xs>, <Xt>, [Xn|SP]

64-bit LDUMAX (size == 11 & A == 1 & R == 0)

LDUMAXA <Xs>, <Xt>, [Xn|SP]

64-bit LDUMAXAL (size == 11 & A == 1 & R == 1)

LDUMAXAL <Xs>, <Xt>, [Xn|SP]

64-bit LDUMAXL (size == 11 & A == 0 & R == 1)

LDUMAXL <Xs>, <Xt>, [Xn|SP]

if !HaveAtomicExt() then UNDEFINED;
integer t = Uint(Rt);
integer n = Uint(Rn);
integer s = Uint(Rs);

integer datasize = 8 << Uint(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' & Rt != '11111' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAX, STUMAXL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD  result = data + value;
    when MemAtomicOp_BIC  result = data AND NOT(value);
    when MemAtomicOp_EOR  result = data EOR value;
    when MemAtomicOp_ORR  result = data OR value;
    when MemAtomicOp_SMAX  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN  result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX  result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN  result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMAXB, STUMAXLB.

Integer
(ARMv8.1)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 1 0 0 0 A R 1 Rs 0 1 1 0 0 0 Rn Rt

size opc

LDUMAXB (A == 1 && R == 0)

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

LDUMAXALB (A == 1 && R == 1)

LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]

LDUMAXB (A == 0 && R == 0)

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

LDUMAXLB (A == 0 && R == 1)

LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAXB, STUMAXLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

bits(64) address;
broadcast(datasize) value;
broadcast(datasize) value;
broadcast(datasize) result;

if HaveMTEExt () then
  SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
  CheckSPA1ignment();
  address = SP[];
else
  address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
  when MemAtomicOp_ADD result = data + value;
  when MemAtomicOp_BIC result = data AND NOT(value);
  when MemAtomicOp_EOR result = data EOR value;
  when MemAtomicOp_ORR result = data OR value;
  when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
  when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
  when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
  when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
  X[t] = ZeroExtend(data, regsize);

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMAXH, STUMAXLH.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 0  | 0  | Rn | Rt |

size opc

LDUMAXH (A == 1 && R == 0)

LDUMAXAH <Ws>, <Wt>, [<Xn|SP]>

LDUMAXALH (A == 1 && R == 1)

LDUMAXALH <Ws>, <Wt>, [<Xn|SP]>

LDUMAXH (A == 0 && R == 0)

LDUMAXH <Ws>, <Wt>, [<Xn|SP]>

LDUMAXLH (A == 0 && R == 1)

LDUMAXLH <Ws>, <Wt>, [<Xn|SP]>

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORM;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAXH, STUMAXLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bids(datasize) value;
bids(datasize) data;
bids(datasize) result;

if HaveMTEExt () then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD   result = data + value;
    when MemAtomicOp_BIC   result = data AND NOT(value);
    when MemAtomicOp_EOR   result = data EOR value;
    when MemAtomicOp_ORR   result = data OR value;
    when MemAtomicOp_SMAX  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN  result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX  result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN  result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMINA and LDUMINAL load from memory with acquire semantics.
- LDUMINL and LDUMINAL store to memory with release semantics.
- LDUMIN has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMIN, STUMINL.

Integer
(ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R1</td>
<td>Rs</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size opc
32-bit LDUMIN (size == 10 & A == 0 & R == 0)

LDUMIN <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMINA (size == 10 & A == 1 & R == 0)

LDUMINA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMINAL (size == 10 & A == 1 & R == 1)

LDUMINAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMINL (size == 10 & A == 0 & R == 1)

LDUMINL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDUMIN (size == 11 & A == 0 & R == 0)

LDUMIN <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINA (size == 11 & A == 1 & R == 0)

LDUMINA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINAL (size == 11 & A == 1 & R == 1)

LDUMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINL (size == 11 & A == 0 & R == 1)

LDUMINL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMIN, STUMINL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD  result = data + value;
    when MemAtomicOp_BIC  result = data AND NOT(value);
    when MemAtomicOp_EOR  result = data EOR value;
    when MemAtomicOp_ORR  result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

    Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAB and LDUMINALB load from memory with acquire semantics.
- LDUMINLB and LDUMINALB store to memory with release semantics.
- LDUMINB has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMINB, STUMINL.

**Integer (ARMv8.1)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LDUMINAB** (A == 1 & R == 0)

LDUMINAB <Ws>, <Wt>, [<Xn|SP>]

**LDUMINALB** (A == 1 & R == 1)

LDUMINALB <Ws>, <Wt>, [<Xn|SP>]

**LDUMINB** (A == 0 & R == 0)

LDUMINB <Ws>, <Wt>, [<Xn|SP>]

**LDUMINLB** (A == 0 & R == 1)

LDUMINLB <Ws>, <Wt>, [<Xn|SP>]

```java
if !HaveAtomicExt() then UNDEFINED;
integer t = Uint(Rt);
integer n = Uint(Rn);
integer s = Uint(Rs);

integer datasize = 8 << Uint(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;

case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMINB, STUMINLB</td>
<td>A == '0' &amp; Rt == '1111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAInAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;
if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH**

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- LDUMINH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias STUMINH, STUMINLH.

### Integer (ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**size** **opc**

**LDUMINAH (A == 1 & R == 0)**

LDUMINAH <Ws>, <Wt>, [<Xn]|SP|]

**LDUMINALH (A == 1 & R == 1)**

LDUMINALH <Ws>, <Wt>, [<Xn]|SP|]

**LDUMINH (A == 0 & R == 0)**

LDUMINH <Ws>, <Wt>, [<Xn]|SP|]

**LDUMINLH (A == 0 & R == 1)**

LDUMINLH <Ws>, <Wt>, [<Xn]|SP|]

```plaintext
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' & R == '11111' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp.ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

**Assembler Symbols**

- <Ws> is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMINH, STUMINLH</td>
<td>$A = '0' &amp; &amp; Rt = '11111'$</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_EOR result = data EOR value;
    when MemAtomicOp_ORR result = data OR value;
    when MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

    Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>1</th>
<th>0</th>
<th>imm9</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

8-bit (size == 00 && opc == 01)

LDUR <Bt>, [<Xn|SP>{, #<simm>}

16-bit (size == 01 && opc == 01)

LDUR <Ht>, [<Xn|SP>{, #<simm>}

32-bit (size == 10 && opc == 01)

LDUR <St>, [<Xn|SP>{, #<simm>}

64-bit (size == 11 && opc == 01)

LDUR <Dt>, [<Xn|SP>{, #<simm>}

128-bit (size == 00 && opc == 11)

LDUR <Qt>, [<Xn|SP>{, #<simm>}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = Uint(Rn);
integer t = Uint(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAisignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>32-bit (size == 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDUR &lt;Wt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit (size == 11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDUR &lt;Xt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

### Assembler Symbols

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
  if opc<0> == '1' then UNDEFINED;
else
  // sign-extending load
  memop = MemOp_LOAD;
  if size == '10' && opc<0> == '1' then UNDEFINED;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;
integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

Unscaled offset

LDURB <Wt>, [<Xn|SP>, {}, #<simm>]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType accctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
if HaveMTEExt() then 
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    bits(64) address;
    bits(datasize) data;
    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if memop == MemOp LOAD && wback && n == t && n != 31 then 
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp STORE && wback && n == t && n != 31 then 
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then 
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[ ];
    else
        address = X[n];

    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp STORE
            if rt_unknown then 
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
        when MemOp LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[ ] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

Unscaled offset

LDURH <Wt>, [<Xn|SP>], #<simm>

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType accctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[ ];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[ ] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

imm9  0  0  Rn  Rt

32-bit (opc == 11)

LDURSB <Wt>, [<Xn|SP>{, #<simm>}

64-bit (opc == 10)

LDURSB <Xt>, [<Xn|SP>{, #<simm>}

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;  // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = $SP[];
else
    address = $X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = $X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            $X[t] = SignExtend(data, regsize);
        else
            $X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        $SP[] = address;
    else
        $X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

<table>
<thead>
<tr>
<th>32-bit (opc == 11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDURSH <code>&lt;Wt&gt;</code>, [&lt;Xn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit (opc == 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDURSH <code>&lt;Xt&gt;</code>, [&lt;Xn</td>
</tr>
</tbody>
</table>

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>`: Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`: Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```collectible
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType accctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOpSTORE, MemOpLOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);
bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOpLOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(UnpredictableWBOVERLAPLD);
    assert c IN {ConstraintWBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when ConstraintWBSUPPRESS wback = FALSE; // writeback is suppressed
        when ConstraintUNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when ConstraintUNDEF UNDEFINED;
        when ConstraintNOP EndOfInstruction();

if memop == MemOpSTORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(UnpredictableWBOVERLAPST);
    assert c IN {ConstraintNONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when ConstraintNONE rt_unknown = FALSE;  // value stored is original value
        when ConstraintUNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when ConstraintUNDEF UNDEFINED;
        when ConstraintNOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOpSTORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOpLOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | imm9| 0  | 0  | Rn |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Unscaled offset**

LDURSW `<Xt>, [<Xn|SP]$, #<simm>`

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp LOAD else MemOp STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);  

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
    when Constraint UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
    when Constraint UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
      Mem[address, datasize DIV 8, acctype] = data;
  when MemOp LOAD
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
      Prefetch(address, t<4:0>);
  when MemOp_PREFETCH
    if wb_unknown then
      address = bits(64) UNKNOWN;
    elseif postindex then
      address = address + offset;
    if n == 31 then
      SP[] = address;
    else
      X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

32-bit (sz == 0)

LDXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

LDXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);
endif memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN then rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF then UNDEFINED;
    when Constraint_NOP then EndOfInstruction();
endif memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN then rt_unknown = TRUE; // store UNKNOWN value
      when Constraint_NONE then rt_unknown = FALSE; // store original value
      when Constraint_UNDEF then UNDEFINED;
      when Constraint_NOP then EndOfInstruction();
  endif s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN then rn_unknown = TRUE; // address is UNKNOWN
      when Constraint_NONE then rn_unknown = FALSE; // address is original base
      when Constraint_UNDEF then UNDEFINED;
      when Constraint_NOP then EndOfInstruction();
  endif n == 31 then
    CheckSPAlignment();
    address = SP[];
  elsif rn_unknown then
    address = bits(64) UNKNOWN;
  else
    address = X[n];
  endif memop of
    when MemOp STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian() then el1 : el2 else el2 : el1;
      else
        data = X[t];
      endif
      bit status = '1';
      // Check whether the Exclusives monitors are set to include the
      // physical memory locations corresponding to virtual address
      // range [address, address+dbytes-1].
      if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);
      endif
    when MemOp LOAD
      // Tell the Exclusives monitors to record a sequence of one or more atomic
      // memory reads from virtual address range [address, address+dbytes-1].
      // The Exclusives monitor will only be set if all the reads are from the
if pair then
  if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
  elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, acctype];
    if BigEndian() then
      X[t] = data<datasize-1:elsize>;
      X[t2] = data<elsize-1:0>;
    else
      X[t] = data<elsize-1:0>;
      X[t2] = data<datasize-1:elsize>;
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
      X[t] = Mem[address + 0, 8, acctype];
      X[t2] = Mem[address + 8, 8, acctype];
    else
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtend(data, regsize);
  end
else // rt unknown
  if dbytes != 0 then
    AArch64.SetExclusiveMonitors(address, dbytes);
  end
end

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

32-bit (size == 10)

LDXR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

LDXR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation
bits(64) address;
bhits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE;  // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE;  // store UNKNOWN value
      when Constraint_NONE rt_unknown = FALSE;    // store original value
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rn_unknown = TRUE;  // address is UNKNOWN
      when Constraint_NONE rn_unknown = FALSE;    // address is original base
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

case memop of
  when MemOp STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    elsif pair then
      bits(datasize DIV 2) el1 = X[t];
      bits(datasize DIV 2) el2 = X[t2];
      data = if BigEndian() then el1 : el2 else el2 : el1;
    else
      data = X[t];
      bit status = '1';
      // Check whether the Exclusives monitors are set to include the
      // physical memory locations corresponding to virtual address
      // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
      // This atomic write will be rejected if it does not refer
      // to the same physical locations after address translation.
      Mem[address, dbytes, acctype] = data;
      status = ExclusiveMonitorsStatus();
      X[s] = ZeroExtend(status, 32);

  when MemOp LOAD
    // Tell the Exclusives monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
  if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
  elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, acctype];
    if BigEndian() then
      X[t] = data<datasize-1:elsize>;
      X[t2] = data<elsize-1:0>;
    else
      X[t] = data<elsize-1:0>;
      X[t2] = data<datasize-1:elsize>;
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
      X[t] = Mem[address + 0, 8, acctype];
      X[t2] = Mem[address + 8, 8, acctype];
    else
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtend(data, regsize);
  end

Operational information
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 1   | 1   | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 1   | 0   | 0   | 0   | 1   |

size L Rs o0 Rt2

No offset

LDXRB <Wt>, [<Xn|SP>{,0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;

Assembler Symbols

<Wt>  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE;  // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE;  // store UNKNOWN value
      when Constraint_NONE rt_unknown = FALSE;  // store original value
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rn_unknown = TRUE;  // address is UNKNOWN
      when Constraint_NONE rn_unknown = FALSE;  // address is original base
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

case memop of
  when MemOp STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    elsif pair then
      bits(datasize DIV 2) el1 = X[t];
      bits(datasize DIV 2) el2 = X[t2];
      data = if BigEndian() then el1 : el2 else el2 : el1;
    else
      data = X[t];
  bit status = '1';
  // Check whether the Exclusives monitors are set to include the
  // physical memory locations corresponding to virtual address
  // range [address, address+dbytes-1].
  if AArch64ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, acctype] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);
  when MemOp LOAD
    // Tell the Exclusives monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
  if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
  elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, acctype];
    if BigEndian() then
      X[t] = data<datasize-1:elsize>;
      X[t2] = data<elsize-1:0>;
    else
      X[t] = data<elsize-1:0>;
      X[t2] = data<datasize-1:elsize>;
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
      X[t] = Mem[address + 0, 8, acctype];
      X[t2] = Mem[address + 8, 8, acctype];
    else
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtend(data, regsize);
  end

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

### No offset

\[
\text{LDXRH} \ <\text{Wt}>\ , \ [<\text{Xn}\mid\text{SP}>\{,\#0\}]
\]

integer \( n = \text{UInt}(\text{Rn}) \);  
integer \( t = \text{UInt}(\text{Rt}) \);  
integer \( t2 = \text{UInt}(\text{Rt2}) \);  // ignored by load/store single register  
integer \( s = \text{UInt}(\text{Rs}) \);  // ignored by all loads and store-release  

\[
\text{AccType} \ \text{acctype} = \text{if} \ o0 = '1' \ \text{then} \ \text{AccType ORDEREDATOMIC} \ \text{else} \ \text{AccType ATOMIC};
\]

boolean \( \text{pair} = \text{FALSE} \);  
\[
\text{MemOp} \ \text{memop} = \text{if} \ L = '1' \ \text{then} \ \text{MemOp LOAD} \ \text{else} \ \text{MemOp STORE};
\]

integer \( \text{elsize} = 8 \times \text{UInt}(\text{size}) \);  
integer \( \text{regsize} = \text{if} \ \text{elsize} = 64 \ \text{then} 64 \ \text{else} 32; \)

integer \( \text{datasize} = \text{if} \ \text{pair} \ \text{then} \ \text{elsize} \times 2 \ \text{else} \ \text{elsize}; \)

### Assembler Symbols

\<\text{Wt}\> \quad \text{Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.}\n
\<\text{Xn}\mid\text{SP}\> \quad \text{Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.}\n
bits(64) address;
bv(bits(datasize) data);
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint UNDEF UNDEFINED;
    when Constraint NOP EndOfInstruction();

if memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP};
    case c of
      when Constraint UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
      when Constraint NONE rt_unknown = FALSE; // store original value
      when Constraint UNDEF UNDEFINED;
      when Constraint NOP EndOfInstruction();

if s == n && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
    when Constraint NONE rn_unknown = FALSE; // address is original base
    when Constraint UNDEF UNDEFINED;
    when Constraint NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

case memop of
  when MemOp STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    elseif pair then
      bits(datasize DIV 2) el1 = X[t];
      bits(datasize DIV 2) el2 = X[t2];
      data = if BigEndian() then el1 : el2 else el2 : el1;
    else
      data = X[t];
  bit status = '1';
  // Check whether the Exclusives monitors are set to include the
  // physical memory locations corresponding to virtual address
  // range [address, address+dbytes-1].
  if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, acctype] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);
  when MemOp LOAD
    // Tell the Exclusives monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
  // ConstranedUNPREDICTABLE case
  X[t] = bits(datasize) UNKNOWN;
elsif elsize == 32 then
  // 32-bit load exclusive pair (atomic)
  data = Mem[address, dbytes, acctype];
  if BigEndian() then
    X[t] = data<datasize-1:elsize>;
    X[t2] = data<elsize-1:0>;
  else
    X[t] = data<elsize-1:0>;
    X[t2] = data<datasize-1:elsize>;
else // elsize == 64
  // 64-bit load exclusive pair (not atomic),
  // but must be 128-bit aligned
  if address != Align(address, dbytes) then
    iswrite = FALSE;
    secondstage = FALSE;
    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
    X[t] = Mem[address + 0, 8, acctype];
    X[t2] = Mem[address + 8, 8, acctype];
  else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This is an alias of LSLV. This means:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode for this instruction.

32-bit (sf == 0)

\[
\text{LSL} \ <Wd>, \ <Wn>, \ <Wm> \\
\]

is equivalent to

\[
\text{LSLV} \ <Wd>, \ <Wn>, \ <Wm> \\
\]

and is always the preferred disassembly.

64-bit (sf == 1)

\[
\text{LSL} \ <Xd>, \ <Xn>, \ < Xm> \\
\]

is equivalent to

\[
\text{LSLV} \ <Xd>, \ <Xn>, \ < Xm> \\
\]

and is always the preferred disassembly.

Assembler Symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSLV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the
destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>l= x11111</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

32-bit (sf == 0 & N == 0 & imms != 011111)

LSL <Wd>, <Wn>, #<shift>

is equivalent to

UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)

and is the preferred disassembly when imms + 1 == immr.

64-bit (sf == 1 & N == 1 & imms != 111111)

LSL <Xd>, <Xn>, #<shift>

is equivalent to

UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)

and is the preferred disassembly when imms + 1 == immr.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<shift> For the 32-bit variant: is the shift amount, in the range 0 to 31.

For the 4-bit variant: is the shift amount, in the range 0 to 63.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias `LSL (register)`.

### 32-bit (sf == 0)

\[
\text{LSLV} <\text{Wd}>, <\text{Wn}>, <\text{Wm}>
\]

### 64-bit (sf == 1)

\[
\text{LSLV} <\text{Xd}>, <\text{Xn}>, <\text{Xm}>
\]

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

### Operation

```plaintext
bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

### Operational information

If `PSTATE.DIT` is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of LSRV. This means:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

\[
\text{LSR }<Wd>, <Wn>, <Wm>
\]

is equivalent to

\[
\text{LSRV }<Wd>, <Wn>, <Wm>
\]

and is always the preferred disassembly.

64-bit (sf == 1)

\[
\text{LSR }<Xd>, <Xn>, <Xm>
\]

is equivalent to

\[
\text{LSRV }<Xd>, <Xn>, <Xm>
\]

and is always the preferred disassembly.

Assembler Symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSRV gives the operational pseudocode for this instruction.

Operational information

If \(\text{PSTATE.DIT} = 1\):

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>x</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 & N == 0 & imms == 011111)

LSR <Wd>, <Wn>, #<shift>

is equivalent to

UBFM <Wd>, <Wn>, #<shift>, #31

and is always the preferred disassembly.

64-bit (sf == 1 & N == 1 & imms == 111111)

LSR <Xd>, <Xn>, #<shift>

is equivalent to

UBFM <Xd>, <Xn>, #<shift>, #63

and is always the preferred disassembly.

Assembler Symbols

- <Wd> is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <shift> For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register).

| sf | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | op2 |
|----|---|---|---|---|---|---|---|---|---|---|---|----|
| Rm | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | Rd |

32-bit (sf == 0)

LSRV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

LSRV <Xd>, <Xn>, <Xm>

| integer d = UInt(Rd); |
| integer n = UInt(Rn); |
| integer m = UInt(Rm); |
| integer datasize = if sf == '1' then 64 else 32; |
| ShiftType shift_type = DecodeShift(op2); |

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias \texttt{MUL}.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>Ra</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

\texttt{MADD <Wd>, <Wn>, <Wm>, <Wa>}

64-bit (sf == 1)

\texttt{MADD <Xd>, <Xn>, <Xm>, <Xa>}

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
integer datasize = destsize;
boolean sub_op = (o0 == '1');
```

Assembler Symbols

- \texttt{<Wd>} is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \texttt{<Wn>} is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \texttt{<Wm>} is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- \texttt{<Wa>} is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.
- \texttt{<Xd>} is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \texttt{<Xn>} is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \texttt{<Xm>} is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- \texttt{<Xa>} is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{MUL}</td>
<td>Ra == '1111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;
if sub_op then
    result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
else
    result = UInt(operand3) + (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 1 1 1 | size | L  M Rm | 0 0 0 0 | H 0 | Rn | Rd | 0 2
```

### Vector

MLA `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<Ts>[<index>]`

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M);  Rmhi = '0';
  when '10' index = UInt(H:L);    Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

### Assembler Symbols

**<Vd>**  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>**  Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>**  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>**  Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size `<Ts>` is H.

**<Ts>**  Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<index>**  Is the element index, encoded in “size:L:H:M”: 

MLA (by element)  Page 865
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 0 1 1 1 1 0 size 1 Rm 1 0 0 1 0 1 Rn Rd
```

Three registers of the same type

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size == '11' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datashize = if Q == '1' then 128 else 64;
- integer elements = datashize DIV esize;
- boolean sub_op = (U == '1');

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in “size:Q”:
  - size Q <T>
    - 00 0 8B
    - 00 1 16B
    - 01 0 4H
    - 01 1 8H
    - 10 0 2S
    - 10 1 4S
    - 11 x RESERVED
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datashize) operand1 = V[n];
bit(datashize) operand2 = V[m];
bit(datashize) operand3 = V[d];
bit(datashize) result;
bit(esize) element1;
bit(esize) element2;
bit(esize) product;
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  product = (UInt(element1) * UInt(element2))<esize-1:0>;
  if sub_op then
    Elem[result, e, esize] = Elem[operand3, e, esize] - product;
  else
    Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```
**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

\[ \text{MLS} \langle \text{Vd} \rangle . \langle \text{T} \rangle , \langle \text{Vn} \rangle . \langle \text{T} \rangle , \langle \text{Vm} \rangle . \langle \text{Ts} \rangle [\langle \text{index} \rangle ] \]

```plaintext
integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

- **<Vd>**
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<T>**
  - Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>**
  - Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- **<Vm>**
  - Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

- **<Ts>**
  - Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<index>**
  - Is the element index, encoded in "size:L:H:M":

---

MLS (by element)
Operation

```c
CheckFPAdvSIMEabled64();
bias(datasize) operand1 = V[n];
bias(idxdsise) operand2 = V[m];
bias(datasize) operand3 = V[d];
bias(datasize) result;
integer element1;
integer element2;
bias(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MLS (vector)**

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | 1   | Rm | 1  | 0  | 0  | 1  | 0  | 1   | Rn | Rd |

Three registers of the same type

**MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

- Integer \( d = \text{UInt}(\text{Rd}) \)
- Integer \( n = \text{UInt}(\text{Rn}) \)
- Integer \( m = \text{UInt}(\text{Rm}) \)
- If size == '11' then UNDEFINED
- Integer \( \text{esize} = 8 \ll \text{UInt(size)} \)
- Integer \( \text{datasize} = \text{if Q == '1' then 128 else 64} \)
- Integer \( \text{elements} = \text{datasize} \div \text{esize} \)
- Boolean \( \text{sub_op} = (U == '1') \)

**Assembler Symbols**

- \(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<T>\) Is an arrangement specifier, encoded in “size-Q”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- \(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  product = (UInt(element1) * UInt(element2))<esize-1:0>;
  if sub_op then
    Elem[result, e, esize] = Elem[operand3, e, esize] - product;
  else
    Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of MSUB. This means:

• The encodings in this description are named to match the encodings of MSUB.
• The description of MSUB gives the operational pseudocode for this instruction.

32-bit (sf == 0)

MNEG <Wd>, <Wn>, <Wm>

is equivalent to

MSUB <Wd>, <Wn>, <Wm>, WZR

and is always the preferred disassembly.

64-bit (sf == 1)

MNEG <Xd>, <Xn>, < Xm>

is equivalent to

MSUB <Xd>, <Xn>, < Xm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
MOV (to/from SP)

Move between register and stack pointer: Rd = Rn.

This is an alias of **ADD (immediate)**. This means:

- The encodings in this description are named to match the encodings of **ADD (immediate)**.
- The description of **ADD (immediate)** gives the operational pseudocode for this instruction.

32-bit (sf == 0)

\[
\begin{array}{ccccccccccccccccc}
 sf & 0 & 0 & 1 & 0 & 0 & 0 & 1 & ! = 1 & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Rn & Rd \\
 op & S & shift & imm12
\end{array}
\]

MOV \(<Wd|WSP>, <Wn|WSP>\)

is equivalent to

ADD \(<Wd|WSP>, <Wn|WSP>, #0\)

and is the preferred disassembly when \([Rd == '11111' || Rn == '11111']\).

64-bit (sf == 1)

MOV \(<Xd|SP>, <Xn|SP>\)

is equivalent to

ADD \(<Xd|SP>, <Xn|SP>, #0\)

and is the preferred disassembly when \([Rd == '11111' || Rn == '11111']\).

**Assembler Symbols**

- \(<Wd|WSP>\) is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \(<Wn|WSP>\) is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- \(<Xd|SP>\) is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \(<Xn|SP>\) is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of **ADD (immediate)** gives the operational pseudocode for this instruction.
**MOV (scalar)**

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of `DUP (element)`. This means:

- The encodings in this description are named to match the encodings of `DUP (element)``
- The description of `DUP (element)` gives the operational pseudocode for this instruction.

### Scalar

```plaintext
MOV <V><d>, <Vn>.<T>[<index>]
```

is equivalent to

```plaintext
DUP <V><d>, <Vn>.<T>[<index>]
```

and is always the preferred disassembly.

### Assembler Symbols

- `<V>` is the destination width specifier, encoded in “imm5”:
  ```
<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>
  ```

- `<d>` is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<T>` is the element width specifier, encoded in “imm5”:
  ```
<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>
  ```

- `<index>` is the element index encoded in “imm5”:
  ```
<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>
  ```

### Operation

The description of `DUP (element)` gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.
MOV (element)

Move vector element to another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero. Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (element). This means:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) gives the operational pseudocode for this instruction.

### Advanced SIMD

$$\text{MOV } <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]$$

is equivalent to

$$\text{INS } <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]$$

and is always the preferred disassembly.

### Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Ts>** Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<index1>** Is the destination element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- **<index2>** Is the source element index encoded in “imm5:imm4”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm4&lt;3:0&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm4&lt;3:1&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm4&lt;3:2&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm4&lt;3&gt;</td>
</tr>
</tbody>
</table>

  Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

### Operation

The description of INS (element) gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
MOV (from general)

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (general). This means:

- The encodings in this description are named to match the encodings of INS (general).
- The description of INS (general) gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>imm5</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Advanced SIMD

MOV <Vd>.<Ts>[<index>], <R><n>

is equivalent to

INS <Vd>.<Ts>[<index>], <R><n>

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> Is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxxx1</td>
<td>W</td>
</tr>
<tr>
<td>xxxx10</td>
<td>W</td>
</tr>
<tr>
<td>xx100</td>
<td>W</td>
</tr>
<tr>
<td>x1000</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

The description of INS (general) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This is an alias of **MOVN**. This means:

- The encodings in this description are named to match the encodings of **MOVN**.
- The description of **MOVN** gives the operational pseudocode for this instruction.

32-bit (sf == 0)

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVN <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when \( \text{!} \left( \text{IsZero}(\text{imm16}) \land \text{hw} \neq '00' \right) \land \text{!} \text{IsOnes}(\text{imm16}) \). 

64-bit (sf == 1)

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVN <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when \( \text{!} \left( \text{IsZero}(\text{imm16}) \land \text{hw} \neq '00' \right) \).

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<imm>` For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xFFFF0000 and 0x0000FFFF
  - For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".
- `<shift>` For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as `<shift>/16`.
  - For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as `<shift>/16`.

Operation

The description of **MOVN** gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.

| sf | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| opc |     |     | hw | imm16 | Rd |

32-bit (sf == 0)

MOV <Wd>, #<imm>

is equivalent to

MOVZ <Wd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00').

64-bit (sf == 1)

MOV <Xd>, #<imm>

is equivalent to

MOVZ <Xd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00').

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<imm> For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw".

For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".

<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (vector)

Move vector. This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of ORR (vector, register). This means:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three registers of the same type

MOV <Vd>.<T>, <Vn>.<T> is equivalent to ORR <Vd>.<T>, <Vn>.<T>, <Vn>.<T> and is the preferred disassembly when \( Rm == Rn \).

Assembler Symbols

- \(<Vd>\) is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<T>\) is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- \(<Vn>\) is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This is an alias of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

32-bit (sf == 0 & N == 0)

MOV <Wd|WSP>, #<imm>

is equivalent to

ORR <Wd|WSP>, WZR, #<imm>

and is the preferred disassembly when \(! MoveWidePreferred(sf, N, imms, immr)\).

64-bit (sf == 1)

MOV <Xd|SP>, #<imm>

is equivalent to

ORR <Xd|SP>, XZR, #<imm>

and is the preferred disassembly when \(! MoveWidePreferred(sf, N, imms, immr)\).

Assembler Symbols

- \(<Wd|WSP>\) Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \(<Xd|SP>\) Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \(<imm>\) For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (register)

Move (register) copies the value in a source register to the destination register.

This is an alias of ORR (shifted register). This means:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

MOV \(<Wd>\), \(<Wm>\)

is equivalent to

ORR \(<Wd>\), WZR, \(<Wm>\)

and is always the preferred disassembly.

64-bit (sf == 1)

MOV \(<Xd>\), \(<Xm>\)

is equivalent to

ORR \(<Xd>\), XZR, \(<Xm>\)

and is always the preferred disassembly.

Assembler Symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Wm>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xm>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (to general)

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of UMOV. This means:

• The encodings in this description are named to match the encodings of UMOV.
• The description of UMOV gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
</tr>
</tbody>
</table>

32-bit (Q == 0 && imm5 == xx100)

MOV <Wd>, <Vn>.S[<index>]

is equivalent to

UMOV <Wd>, <Vn>.S[<index>]

and is always the preferred disassembly.

64-reg,UMOV-64-reg (Q == 1 && imm5 == x1000)

MOV <Xd>, <Vn>.D[<index>]

is equivalent to

UMOV <Xd>, <Vn>.D[<index>]

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<index> For the 32-bit variant: is the element index encoded in "imm5<4:3>".
For the 64-reg,UMOV-64-reg variant: is the element index encoded in "imm5<4:3>".

Operation

The description of UMOV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ▪ The values of the data supplied in any of its registers.
  ▪ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ▪ The values of the data supplied in any of its registers.
  ▪ The values of the NZCV flags.
MOVI

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

8-bit (op == 0 && cmode == 1110)

MOVI <Vd>.<T>, #<imm8>{, LSL #0}

16-bit shifted immediate (op == 0 && cmode == 10x0)

MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}

32-bit shifted immediate (op == 0 && cmode == 0xx0)

MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}

32-bit shifting ones (op == 0 && cmode == 110x)

MOVI <Vd>.<T>, #<imm8>, LSL #<amount>

64-bit scalar (Q == 0 && op == 1 && cmode == 1110)

MOVI <Dd>, #<imm>

64-bit vector (Q == 1 && op == 1 && cmode == 1110)

MOVI <Vd>.2D, #<imm>

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
  when '11111' // FMOV Dn,#imm is in main FP instruction set
  if Q == '0' then UNDEFINED;
  operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
### Assembler Symbols

- `<Dd>`
  - Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Vd>`
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- `<imm>`
  - Is a 64-bit immediate `aaaaaaabbbbbbcccccccccddddddeeeeeeefffffffffgggggggghhhhhhh`, encoded in "a:b:c:d:e:f:g:h:.

- `<T>`
  - For the 8-bit variant: is an arrangement specifier, encoded in “Q”:
    
    | Q | <T> |
    |---|-----|
    | 0 | 8B  |
    | 1 | 16B |

  - For the 16-bit variant: is an arrangement specifier, encoded in “Q”:
    
    | Q | <T> |
    |---|-----|
    | 0 | 4H  |
    | 1 | 8H  |

  - For the 32-bit variant: is an arrangement specifier, encoded in “Q”:
    
    | Q | <T> |
    |---|-----|
    | 0 | 2S  |
    | 1 | 4S  |

- `<imm8>`
  - Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h:

- `<amount>`
  - For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:
    
    | cmode<1> | <amount> |
    |----------|----------|
    | 0        | 0        |
    | 1        | 8        |

  - defaulting to 0 if LSL is omitted.

  - For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:
    
    | cmode<2:1> | <amount> |
    |------------|----------|
    | 00         | 0        |
    | 01         | 8        |
    | 10         | 16       |
    | 11         | 24       |

  - defaulting to 0 if LSL is omitted.

  - For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:
    
    | cmode<0> | <amount> |
    |----------|----------|
    | 0        | 8        |
    | 1        | 16       |

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = Imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);
V[rd] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MOVK <Wd>, #<imm>{, LSL #<shift>}

64-bit (sf == 1)

MOVK <Xd>, #<imm>{, LSL #<shift>}

```plaintext
d = UInt(Rd);
datasize = if sf == '1' then 64 else 32;
imm = imm16;
pos = UInt(hw:'0000');
```

case opc of
  when '00' opcode = MoveWideOp_N;
  when '10' opcode = MoveWideOp_Z;
  when '11' opcode = MoveWideOp_K;
  otherwise UNDEFINED;

if sf == '0' && hw<1> == '1' then UNDEFINED;

Assembler Symbols

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<imm>` is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- `<shift>` is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

Operation

```plaintext
result = X[d];
result<pos+15:pos> = imm;
result = NOT(result);
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
**MOVN**

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias **MOV (inverted wide immediate)**.

32-bit (sf == 0)

MOVN <Wd>, #<imm>{, LSL #<shift>}

64-bit (sf == 1)

MOVN <Xd>, #<imm>{, LSL #<shift>}

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
bits(16) imm = imm16;
integer pos;
MoveWideOp opcode;

case opc of
   when '00' opcode = MoveWideOp_N;
   when '10' opcode = MoveWideOp_Z;
   when '11' opcode = MoveWideOp_K;
   otherwise UNDEFINED;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

**Assembler Symbols**

- `<Wd>`  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>`  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<imm>`  Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- `<shift>`  For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>64-bit</td>
<td>! (IsZero(imm16) &amp;&amp; hw != '00')</td>
</tr>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>32-bit</td>
<td>! (IsZero(imm16) &amp;&amp; hw != '00') &amp;&amp; ! IsOnes(imm16)</td>
</tr>
</tbody>
</table>
Operation

```c
bits(datasize) result;

if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();
result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
    result = NOT(result);
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (wide immediate).

<table>
<thead>
<tr>
<th>sf</th>
<th>1 0 1 0 0 1 0</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MOVZ <Wd>, #<imm>{, LSL #<shift>}

64-bit (sf == 1)

MOVZ <Xd>, #<imm>{, LSL #<shift>}

integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer imm = imm16;
integer pos;
MoveWideOp opcode;
case opc of
  when '00' opcode = MoveWideOp N;
  when '10' opcode = MoveWideOp Z;
  when '11' opcode = MoveWideOp K;
  otherwise UNDEFINED;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm> Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (wide immediate)</td>
<td>!(IsZero(imm16) &amp;&amp; hw != '00')</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;

if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();
result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
    result = NOT(result);
X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MRS

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | o0 | op1 | CRn | CRm | op2 | Rt | L |

System

MRS <Xt>, (<systemreg>_<op0>_<op1>_<Cn>_<Cm>_<op2>)

AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
boolean read = (L == '1');

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".
The System register names are defined in "AArch64 System Registers" in the System Register XML.

<op0> Is an unsigned immediate, encoded in "o0":

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

if read then
   X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
   AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see Process state. PSTATE.

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If ARMv8.0-SSBS is implemented, PSTATE.SSBS.
- If ARMv8.1-PAN is implemented, PSTATE.PAN.
- If ARMv8.2-UAO is implemented, PSTATE.UAO.
- If ARMv8.4-DIT is implemented, PSTATE.DIT.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | CRm | 0  | 0  | 0  | 0  | op1 | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  |

System

MSR <pstatefield>, #<imm>

```sysasm
if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFlag";
if op1 == '000' && op2 == '010' then SEE "AXFlag";
AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');

bits(4) operand = CRm;
PSTATEField field;
case op1:op2 of
  when '000 011' if !HaveUAOExt() then UNDEFINED;
    field = PSTATEField_UAO;
  when '000 100' if !HavePANExt() then UNDEFINED;
    field = PSTATEField_PAN;
  when '000 101' field = PSTATEField_SP;
  when '011 010' if !HaveDITExt() then UNDEFINED;
    field = PSTATEField_DIT;
  when '011 110' field = PSTATEField_DAIFSet;
  when '011 111' field = PSTATEField_DAIFClr;
  when '011 001' if !HaveSSBSExt() then UNDEFINED;
    field = PSTATEField_SSBS;
  otherwise UNDEFINED;

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if op1 == '011' && PSTATE.EL == EL0 && (IsInHost() || SCTLR_EL1.UMA == '0') then
  AArch64.SystemRegisterTrap(EL1, '00', op2, op1, '0100', '11111', CRm, '0');
```

Assembler Symbols

<pstatefield> Is a PSTATE field name, encoded in “op1:op2”:
<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>&lt;pstatefield&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00x</td>
<td>SEE PSTATE</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>SEE PSTATE</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>UAO</td>
<td>ARMv8.2-UAO</td>
</tr>
<tr>
<td>000</td>
<td>100</td>
<td>PAN</td>
<td>ARMv8.1-PAN</td>
</tr>
<tr>
<td>000</td>
<td>101</td>
<td>SPSel</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>11x</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>xxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>xxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
<td>SSBS</td>
<td>ARMv8.0-SpecRest</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>DIT</td>
<td>ARMv8.4-DIT</td>
</tr>
<tr>
<td>011</td>
<td>011</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>10x</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
<td>DAIFSet</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>111</td>
<td>DAIFClr</td>
<td>-</td>
</tr>
<tr>
<td>1xx</td>
<td>xxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
</tbody>
</table>

*<imm>* Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

**Operation**

```c
case field of
    when PSTATEField_SSBS
        PSTATE.SSBS = operand<0>;
    when PSTATEField_SP
        PSTATE.SP = operand<0>;
    when PSTATEField_DAIFSet
        PSTATE.D = PSTATE.D OR operand<3>;
        PSTATE.A = PSTATE.A OR operand<2>;
        PSTATE.I = PSTATE.I OR operand<1>;
        PSTATE.F = PSTATE.F OR operand<0>;
    when PSTATEField_DAIFClr
        PSTATE.D = PSTATE.D AND NOT(operand<3>);
        PSTATE.A = PSTATE.A AND NOT(operand<2>);
        PSTATE.I = PSTATE.I AND NOT(operand<1>);
        PSTATE.F = PSTATE.F AND NOT(operand<0>);
    when PSTATEField_PAN
        PSTATE.PAN = operand<0>;
    when PSTATEField_UAO
        PSTATE.UAO = operand<0>;
    when PSTATEField_DIT
        PSTATE.DIT = operand<0>;
```
MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | o0 | op1 | CRn | CRm | op2 | Rt |

**System**

MSR (<systemreg>|S<op0>_<op1>_<>cn>_<>cm>_<>op2>), <Xt>

```assembly
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = Uint(Rt);
integer sys_op0 = 2 + Uint(o0);
integer sys_op1 = Uint(op1);
integer sys_op2 = Uint(op2);
integer sys_crn = Uint(CRn);
integer sys_crm = Uint(CRm);
boolean read = (L == '1');
```

**Assembler Symbols**

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".
The System register names are defined in 'AArch64 System Registers' in the System Register XML.

<op0> Is an unsigned immediate, encoded in "o0":

<table>
<thead>
<tr>
<th>o0</th>
<th>&lt;op0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

**Operation**

```assembly
if read then
    X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```
**MSUB**

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias `MNEG`.

32-bit (sf == 0)

```plaintext
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit (sf == 1)

```plaintext
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
integer datasize = destsize;
boolean sub_op = (o0 == '1');

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Wa>` Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Xa>` Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MNEG</code></td>
<td><code>Ra == '11111'</code></td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;
if sub_op then
    result = UInt(operand3) - (UInt(operand1) *UInt(operand2));
else
    result = UInt(operand3) + (UInt(operand1) *UInt(operand2));
X[d] = result<destsize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the \texttt{CPACR_EL1}, \texttt{CPTR_EL2}, and \texttt{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 0  | 1  | 1  | 1  | L  | M  | Rm | 1  | 0  | 0  | H  | 0  | Rn | Rd |

Vector

\[ \text{MUL } <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>] \]

\[
\begin{align*}
\text{integer idxdsize} & = \text{if } H == '1' \text{ then } 128 \text{ else } 64; \\
\text{integer index;} & \\
\text{bit Rmhi;} & \\
\text{case size of} & \\
\text{when } '01' \text{ index} & = \text{UInt}(H:L:M); \ Rmhi = '0'; \\
\text{when } '10' \text{ index} & = \text{UInt}(H:L); \ Rmhi = M; \\
\text{otherwise UNDEFINED;} & \\
\text{integer d} & = \text{UInt}(Rd); \\
\text{integer n} & = \text{UInt}(Rn); \\
\text{integer m} & = \text{UInt}(Rmhi:Rm); \\
\text{integer esize} & = 8 << \text{UInt}(size); \\
\text{integer datasize} & = \text{if } Q == '1' \text{ then } 128 \text{ else } 64; \\
\text{integer elements} & = \text{datasize DIV esize};
\end{align*}
\]

Assembler Symbols

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

\(<Ts>\) Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<\text{index}>\) Is the element index, encoded in "size:L:H:M":

MUL (by element)
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = \text{V}[n];
bits(idxdsize) operand2 = \text{V}[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

\text{element2} = \text{UInt}(\text{Elem}[\text{operand2, index, esize}]);
\text{for} \ e = 0 \ \text{to} \ \text{elements}-1
\text{element1} = \text{UInt}(\text{Elem}[\text{operand1, e, esize}]);
\text{product} = (\text{element1} \ \ast \ \text{element2})<\text{esize}-1:0>;
\text{Elem}[\text{result, e, esize}] = \text{product};
\text{V}[d] = \text{result};

Operational information

If PSTATE.DIT is 1:
\begin{itemize}
\item The execution time of this instruction is independent of:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\item The response of this instruction to asynchronous exceptions does not vary based on:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\end{itemize}
MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

MUL <Vd> <T>, <Vn> <T>, <Vm> <T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean poly = (U == '1');
```

Assembler Symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1) * UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MUL

Multiply: \( Rd = Rn \times Rm \)

This is an alias of MADD. This means:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 0 1 1 0 0</th>
<th>Rm</th>
<th>0 1 1 1 1 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td></td>
<td>Ra</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MUL <Wd>, <Wn>, <Wm>

is equivalent to

MADD <Wd>, <Wn>, <Wm>, WZR.

and is always the preferred disassembly.

64-bit (sf == 1)

MUL <Xd>, <Xn>, <Xm>

is equivalent to

MADD <Xd>, <Xn>, <Xm>, XZR.

and is always the preferred disassembly.

Assembler Symbols

- \(<Wd>\): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\): Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \(<Wm>\): Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- \(<Xd>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\): Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \(<Xm>\): Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MADD gives the operational pseudocode for this instruction.
Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of NOT. This means:

- The encodings in this description are named to match the encodings of NOT.
- The description of NOT gives the operational pseudocode for this instruction.

Arrangement:

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector

MVN <Vd>.<T>, <Vn>.<T>

is equivalent to

NOT <Vd>.<T>, <Vn>.<T>

and is always the preferred disassembly.

Assembler Symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th></th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8B</td>
</tr>
<tr>
<td></td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of NOT gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This is an alias of ORN (shifted register). This means:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

\[
\text{MVN } <Wd>, <Wm>\{, <shift> \#<amount>\}
\]

is equivalent to

\[
\text{ORN } <Wd>, WZR, <Wm>\{, <shift> \#<amount>\}
\]

and is always the preferred disassembly.

### 64-bit (sf == 1)

\[
\text{MVN } <Xd>, <Xm>\{, <shift> \#<amount>\}
\]

is equivalent to

\[
\text{ORN } <Xd>, XZR, <Xm>\{, <shift> \#<amount>\}
\]

and is always the preferred disassembly.

### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  - For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

### Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its registers.
The values of the NZCV flags.
MVNI

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### 16-bit shifted immediate (cmode == 10x0)

\[
\text{MVNI } <Vd>,<T>, \#<imm8>, \text{LSL } \#<amount>
\]

### 32-bit shifted immediate (cmode == 0x0)

\[
\text{MVNI } <Vd>,<T>, \#<imm8>, \text{LSL } \#<amount>
\]

### 32-bit shifting ones (cmode == 110x)

\[
\text{MVNI } <Vd>,<T>, \#<imm8>, \text{MSL } \#<amount>
\]

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;

\begin{verbatim}
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = ImmediateOp_MOVI;
\end{verbatim}

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

### Assembler Symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is the arrangement specifier, encoded in "Q":

```
\begin{array}{c}
\text{Q} \\
0 \\
1
\end{array}
\begin{array}{c}
\text{<T>} \\
4H \\
8H
\end{array}
```

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

```
Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

<table>
<thead>
<tr>
<th>cmode&lt;1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

<table>
<thead>
<tr>
<th>cmode&lt;0&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);
V[rd] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**NEG (vector)**

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1, CPTR_EL2, and CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | Rn |  | Rd |

**Scalar**

`NEG <V><d>, <V><n>`

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

**Vector**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | Rn |  | Rd |

**Vector**

`NEG <Vd>.<T>, <Vn>.<T>`

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

**Assembler Symbols**

- **<V>** Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<d>** Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- **<n>** Is the number of the SIMD&FP source register, encoded in the "Rn" field.

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<T>** Is an arrangement specifier, encoded in "size:Q":

Page 913
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{n}> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of SUB (shifted register). This means:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode for this instruction.

![Register File](image)

32-bit (sf == 0)

\[
\text{NEG} \ <Wd>, \ <Wm>\{, \ <shift> \ #<amount>\}
\]

is equivalent to

\[
\text{SUB} \ <Wd>, \ WZR, \ <Wm> \{, \ <shift> \ #<amount>\}
\]

and is always the preferred disassembly.

64-bit (sf == 1)

\[
\text{NEG} \ <Xd>, \ <Xm>\{, \ <shift> \ #<amount>\}
\]

is equivalent to

\[
\text{SUB} \ <Xd>, \ XZR, \ <Xm> \{, \ <shift> \ #<amount>\}
\]

and is always the preferred disassembly.

Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>`: Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>`: Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<shift>`: Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>`: For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.
NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

NEGS \(<Wd>, <Wm>\{, <shift> #<amount>\}

is equivalent to

SUBS \(<Wd>, WZR, <Wm>\{, <shift> #<amount>\}

and is always the preferred disassembly.

64-bit (sf == 1)

NEGS \(<Xd>, <Xm>\{, <shift> #<amount>\}

is equivalent to

SUBS \(<Xd>, XZR, <Xm>\{, <shift> #<amount>\}

and is always the preferred disassembly.

Assembler Symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Wm>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xm>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

\(<shift>\) Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<amount>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of SBC. This means:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rd</th>
</tr>
</thead>
</table>

op S Rn

32-bit (sf == 0)

NGC <Wd>, <Wm>

is equivalent to

SBC <Wd>, WZR, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

NGC <Xd>, < Xm>

is equivalent to

SBC <Xd>, XZR, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-008e8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SBCS. This means:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode for this instruction.

32-bit (sf == 0)

NGCS <Wd>, <Wm>

is equivalent to

SBCS <Wd>, WZR, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

NGCS <Xd>, <Xm>

is equivalent to

SBCS <Xd>, XZR, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

System

NOP

SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 111' see "XPACLRI";
    when '0001 xxx' see "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
    when '0010 000' if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_ESB;
    when '0010 001' if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_PSB;
    when '0010 010' if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_TSB;
    when '0010 100' op = SystemHintOp_CSDB;
    when '0011 xxx' see "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    when '0100 xx0' op = SystemHintOp_BTI;
        BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
    otherwise EndOfInstruction(); // Instruction executes as NOP
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap(EL1, TRUE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap(EL3, TRUE);
      WaitForEvent();
    when SystemHintOp_WFI
      if !InterruptPending() then
        if PSTATE.EL == EL0 then
          // Check for traps described by the OS which may be EL1 or EL2.
          AArch64.CheckForWfxTrap(EL1, FALSE);
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
          // Check for traps described by the Hypervisor.
          AArch64.CheckForWfxTrap(EL2, FALSE);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
          // Check for traps described by the Secure Monitor.
          AArch64.CheckForWfxTrap(EL3, FALSE);
        WaitForInterrupt();
      when SystemHintOp_SEV
        SendEvent();
      when SystemHintOp_SEVL
        SendEventLocal();
      when SystemHintOp_ESB
        SynchronizeErrors();
        AArch64.ESBOperation();
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
          AArch64.vESBOperation();
        TakeUnmaskedSErrorInterrupts();
      when SystemHintOp_PSB
        ProfilingSynchronizationBarrier();
      when SystemHintOp_TSB
        TraceSynchronizationBarrier();
      when SystemHintOp_CSDB
        ConsumptionOfSpeculativeDataBarrier();
      when SystemHintOp_BTI
        BTypeNext = '00';
    otherwise // do nothing
    Operative information
    If PSTATE.DIT is 1:
      • The execution time of this instruction is independent of:
        ◦ The values of the data supplied in any of its registers.
        ◦ The values of the NZCV flags.
      • The response of this instruction to asynchronous exceptions does not vary based on:
        ◦ The values of the data supplied in any of its registers.
        ◦ The values of the NZCV flags.
Not

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias \texttt{MVN}.

\begin{center}
\begin{tabular}{cccccccccccccccccc}
0 & Q & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & Rn & Rr & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & Rr & Rd
\end{tabular}
\end{center}

**Vector**

\[
\text{NOT} <Vd>.<T>, <Vn>.<T>
\]

\[
\text{integer } d = \text{UInt}(Rd);
\]

\[
\text{integer } n = \text{UInt}(Rn);
\]

\[
\text{integer } esize = 8;
\]

\[
\text{integer } datasize = \text{if } Q == '1' \text{ then } 128 \text{ else } 64;
\]

\[
\text{integer } \text{elements} = \text{datasize} \text{ DIV } 8;
\]

**Assembler Symbols**

Air \texttt{<Vd>} \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}

Air \texttt{<T>} \quad \text{Is an arrangement specifier, encoded in “Q”:}

\[
\begin{array}{c|c}
Q & <T> \\
0 & 8B \\
1 & 16B
\end{array}
\]

\texttt{<Vn>} \quad \text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}

**Operation**

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\text{bits(datasize) } \text{operand} = \text{V}[n];
\]

\[
\text{bits(datasize) } \text{result};
\]

\[
\text{bits(esize) } \text{element};
\]

\[
\text{for } e = 0 \text{ to elements-1}
\]

\[
\text{\hspace{1em}element} = \text{Elem}[\text{operand}, e, esize];
\]

\[
\text{\hspace{1em}}\text{Elem}[\text{result}, e, esize] = \text{NOT}(\text{element});
\]

\[
\text{V}[d] = \text{result};
\]

**Operational information**

If PSTATE.DIT is 1:

\begin{itemize}
\item The execution time of this instruction is independent of:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\item The response of this instruction to asynchronous exceptions does not vary based on:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\end{itemize}
ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | |

Three registers of the same type

ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean invert = (size<0> == '1');
LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND
    result = operand1 AND operand2;
  when LogicalOp_ORR
    result = operand1 OR operand2;

V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>shift</th>
<th>1</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

32-bit (sf == 0)

ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVN</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND  result = operand1 AND operand2;
  when LogicalOp_ORR   result = operand1 OR operand2;
  when LogicalOp_EOR   result = operand1 EOR operand2;
if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ORR (vector, immediate)**

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | d  | e  | f  | g  | h  | Rd |

**16-bit (cmode == 10x1)**

ORR `<Vd>.<T>`, #<imm8>{, LSL #<amount>}

**32-bit (cmode == 0x1)**

ORR `<Vd>.<T>`, #<imm8>{, LSL #<amount>}

```plaintext
integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;

case cmode:op of
when '0xx00' operation = ImmediateOp_MOVI;
when '0xx01' operation = ImmediateOp_MVNI;
when '0xx10' operation = ImmediateOp_ORR;
when '0xx11' operation = ImmediateOp_BIC;
when '10x00' operation = ImmediateOp_MOVI;
when '10x01' operation = ImmediateOp_MVNI;
when '10x10' operation = ImmediateOp_ORR;
when '10x11' operation = ImmediateOp_BIC;
when '110x0' operation = ImmediateOp_MOVI;
when '110x1' operation = ImmediateOp_MVNI;
when '1110x' operation = ImmediateOp_MOVI;
when '11110' operation = ImmediateOp_MOVI;
when '11111' // FMOV Dn,#imm is in main FP instruction set
    // if Q == '0' then UNDEFINED;
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP register, encoded in the "Rd" field.
- `<T>` For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

- `<imm8>` Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".
- `<amount>` For the 16-bit variant: is the shift amount encoded in “cmode<1>”:
For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);
V[rd] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ORR (vector, register)**

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias `MOV (vector)`.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | Rm | 0  | 0  | 0  | 1  | 1  | Rd | 0  | 0  | 1  | 1  | Rn | 0  | 0  | 1  | 1  | Rd | 0  | 0  | 1  |

### Three registers of the same type

```
ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean invert = (size<0> == '1');

逻辑运算符 op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector)</td>
<td>Rm == Rn</td>
</tr>
</tbody>
</table>

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
if invert then operand2 = NOT(operand2);
case op of
  when LogicalOp_AND
    result = operand1 AND operand2;
  when LogicalOp_ORR
    result = operand1 OR operand2;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ORR (immediate)**

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate).

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit \((sf == 0 && N == 0)\)

ORR \(<Wd|WSP>, <Wn>, \#<imm>\)

64-bit \((sf == 1)\)

ORR \(<Xd|SP>, <Xn>, \#<imm>\)

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

**Assembler Symbols**

\(<Wd|WSP>\) Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

\(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

\(<Xd|SP>\) Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

\(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

\(<imm>\) For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (bitmask immediate)</td>
<td>Rn == '11111' &amp;&amp; !MoveWidePreferred(sf, N, imms, immr)</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;
bits(datasize) operand1 = \text{X}[n];
bits(datasize) operand2 = imm;

case op of
  when \text{LogicalOp\ AND} result = operand1 \text{ AND} operand2;
  when \text{LogicalOp\ ORR} result = operand1 \text{ OR} operand2;
  when \text{LogicalOp\ EOR} result = operand1 \text{ EOR} operand2;

if setflags then
  \text{PSTATE}.<N,Z,C,V> = result<\text{datasize}-1>:\text{IsZeroBit}(result):'00';
if d == 31 && !setflags then
  \text{SP}[] = result;
else
  \text{X}[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register).

### 32-bit (sf == 0)

```plaintext
ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```plaintext
ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');

### Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>`: Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:
  ```plaintext
  shift  <shift>
  00   LSL
  01   LSR
  10   ASR
  11   ROR
  ```
  - For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  - For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (register)</td>
<td>( \text{shift} = '00' &amp;&amp; \text{imm6} = '000000' &amp;&amp; \text{Rn} = '11111' )</td>
</tr>
</tbody>
</table>

**Operation**

\[
\begin{align*}
\text{bits}(\text{datasize}) \ \text{operand1} &= X[n]; \\
\text{bits}(\text{datasize}) \ \text{operand2} &= \text{ShiftReg}(m, \text{shift\_type}, \text{shift\_amount}); \\
\text{if invert then operand2} &= \text{NOT}(\text{operand2}); \\
\text{case op of} \\
\text{when LogicalOp\_AND} \ \text{result} &= \text{operand1 \ AND \ operand2}; \\
\text{when LogicalOp\_ORR} \ \text{result} &= \text{operand1 \ OR \ operand2}; \\
\text{when LogicalOp\_EOR} \ \text{result} &= \text{operand1 \ EOR \ operand2}; \\
\text{if setflags then} \\
\text{PSTATE.<N,Z,C,V>} &= \text{result<datasize-1>} : \text{IsZeroBit}(\text{result}):'00'; \\
X[d] &= \text{result};
\end{align*}
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**PACDA, PACDZA**

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

### Integer

(ARMv8.3)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |

**PACDA (Z == 0)**

PACDA <Xd>, <Xn|SP>

**PACDZA (Z == 1 && Rn == 11111)**

PACDZA <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UNDEFINED;
```

**Assembler Symbols**

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

**Operation**

```java
if source_is_sp then
    X[d] = AddPACDA(X[d], SP[]);
else
    X[d] = AddPACDA(X[d], X[n]);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDB.
- The value zero, for PACDZB.

### Integer (ARMv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 0  | 1  | 1  |

#### PACDB (Z == 0)

PACDB <Xd>, <Xn|SP>

#### PACDZB (Z == 1 && Rn == 11111)

PACDZB <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UNDEFINED;
```

### Assembler Symbols

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

### Operation

```java
if source_is_sp then
    X[d] = AddPACDB(X[d], SP[]);
else
    X[d] = AddPACDB(X[d], X[n]);
```

---

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

Integer (ARMv8.3)

```
|   31   | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|--------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|  1    | 0   | 0   | 1   | 1   | 0   | 1   | 0   | 1   | 1   | 0   | Rm  | 0   | 0   | 1   | 1   | 0   | 0   | Rn  | Rd  |
```

PACGA <Xd>, <Xn>, <Xm|SP>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !HavePACExt() then
  UNDEFINED;

if m == 31 then source_is_sp = TRUE;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

```plaintext
if source_is_sp then
  X[d] = AddPACGA(X[n], SP());
else
  X[d] = AddPACGA(X[n], X[m]);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:
• In the general-purpose register that is specified by <Xd> for PACIA and PACIZA.
• In X17, for PACIA1716.
• In X30, for PACIASP and PACIAZ.

The modifier is:
• In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIA.
• The value zero, for PACIZA and PACIAZ.
• In X16, for PACIA1716.
• In SP, for PACIASP.

It has encodings from 2 classes: Integer and System.

Integer
(ARMv8.3)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PACIA (Z == 0)

PACIA <Xd>, <Xn|SP>

PACIZA (Z == 1 && Rd == 11111)

PACIZA <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
  UNDEFINED;

if Z == '0' then // PACIA
  if n == 31 then source_is_sp = TRUE;
  else // PACIZA
    if n != 31 then UNDEFINED;
```

System
(ARMv8.3)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CRm</td>
<td>op2</td>
<td></td>
</tr>
</tbody>
</table>
PACIA1716 (CRm == 0001 && op2 == 000)

PACIA1716

PACIASP (CRm == 0011 && op2 == 001)

PACIASP

PACIAZ (CRm == 0011 && op2 == 000)

PACIAZ

integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 000' // PACIAZ
        d = 30;
        n = 31;
    when '0011 001' // PACIASP
        d = 30;
        source_is_sp = TRUE;
        if HaveBTIExt () then
            BTypeCompatible = BTypeCompatible_PACIXSP();
    when '0001 000' // PACIA1716
        d = 17;
        n = 16;
    when '0001 010' SEE "PACIB";
    when '0001 100' SEE "AUTIA";
    when '0001 110' SEE "AUTIB";
    when '0011 01x' SEE "PACIB";
    when '0011 10x' SEE "AUTIA";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if HavePACExt () then
    if source_is_sp then
        X[d] = AddPACIA(X[d], SP[]);
    else
        X[d] = AddPACIA(X[d], X[n]);
PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:
- In the general-purpose register that is specified by <Xd> for PACIB and PACIZB.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIB.
- The value zero, for PACIZB and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

It has encodings from 2 classes: Integer and System

### Integer

(ARMv8.3)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### PACIB (Z == 0)

PACIB <Xd>, <Xn|SP>

### PACIZB (Z == 1 & & Rn == 1111)

PACIZB <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
  UNDEFINED;

if Z == '0' then // PACIB
  if n == 31 then source_is_sp = TRUE;
else // PACIZB
  if n != 31 then UNDEFINED;
```

### System

(ARMv8.3)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | x  | 1  | 0  | 1  | x  | 1  | 1  | 1  | 1  | 1  | }
PACIB1716 (CRm == 0001 && op2 == 010)

PACIB1716

PACIBSP (CRm == 0011 && op2 == 011)

PACIBSP

PACIBZ (CRm == 0011 && op2 == 010)

PACIBZ

text

integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 010' // PACIBZ
    d = 30;
    n = 31;
  when '0011 011' // PACIBSP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIExt() then
      BTypeCompatible = BTypeCompatible_PACIXSP();
  when '0001 010' // PACIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLRI";

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if HavePACExt() then
  if source_is_sp then
    X[d] = AddPACIB(X[d], SP[]);
  else
    X[d] = AddPACIB(X[d], X[n]);

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00ber8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see Polynomial arithmetic over \{0, 1\}.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | size | 1  | Rm | 1  | 0  | 0  | 1  | 1  | 1  | Rd | Rn | |

Three registers of the same type

\[
\text{PMUL} \ <Vd>,<T>, \ <Vn>,<T>, \ <Vm>,<T> \\
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(\text{Rd}); \\
\text{integer } n &= \text{UInt}(\text{Rn}); \\
\text{integer } m &= \text{UInt}(\text{Rm}); \\
\text{if } U == '1' \&\& \text{ size } != '00' \text{ then UNDEFINED}; \\
\text{if } \text{size } == '11' \text{ then UNDEFINED}; \\
\text{integer } esize &= 8 << \text{UInt}(\text{size}); \\
\text{integer } \text{datasize } &= \text{if } Q == '1' \text{ then } 128 \text{ else } 64; \\
\text{integer } \text{elements } &= \text{datasize} \div \text{esize}; \\
\text{boolean } \text{poly } &= (U == '1'); \\
\end{align*}
\]

Assembler Symbols

\[
\begin{align*}
\text{<Vd>} & \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.} \\
\text{<T>} & \quad \text{Is an arrangement specifier, encoded in "size:Q";} \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{<Vn>} & \quad \text{Is the name of the first SIMD&FP source register, encoded in the "Rn" field.} \\
\text{<Vm>} & \quad \text{Is the name of the second SIMD&FP source register, encoded in the "Rm" field.} \\
\end{align*}
\]

Operation

\[
\begin{align*}
\text{CheckFPAdvSIMDEnabled64}(); \\
\text{bits(datasize) operand1 } &= V[n]; \\
\text{bits(datasize) operand2 } &= V[m]; \\
\text{bits(datasize) result}; \\
\text{bits(esize) element1}; \\
\text{bits(esize) element2}; \\
\text{bits(esize) product}; \\
\text{for } e = 0 \text{ to elements-1} \\
\text{element1 } &= \text{Elem}[\text{operand1}, e, \text{esize}]; \\
\text{element2 } &= \text{Elem}[\text{operand2}, e, \text{esize}]; \\
\text{if poly then} \\
\text{product } &= \text{PolynomialMult}(\text{element1}, \text{element2})<\text{esize}-1:0>; \\
\text{else} \\
\text{product } &= (\text{UInt}(\text{element1}) \times \text{UInt}(\text{element2}))<\text{esize}-1:0>; \\
\text{Elem}[\text{result}, e, \text{esize} ] &= \text{product}; \\
V[d] &= \text{result}; \\
\end{align*}
\]
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
PMULL, PMULL2

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over \{0, 1\}*. 

The PMULL instruction extracts each source vector from the lower half of each source register, while the PMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1, CPTR_EL2, and CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Three registers, not all the same type

\[
\begin{array}{ccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 \\
& & & & & & \\
& & & & & & \\
0 & Q & 0 & 1 & 1 & 0 & \text{size} & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
24 & 23 & 22 & 21 & 20 & 19 & 18 & 17 \\
& & & & & & & \\
& & & & & & & \\
Rm & 1 & 1 & 0 & 0 & Rn & Rd & \\
\end{array}
\]

**Assembler Symbols**

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

\[
\begin{array}{c|c}
Q & 2 \\
\hline
0 & [\text{absent}] \\
1 & [\text{present}] \\
\end{array}
\]

\(<Vd>, <Ta>, <Vn>, <Vm>\)

\[
\begin{array}{ll}
\text{integer } d & = \text{UInt}(Rd); \\
\text{integer } n & = \text{UInt}(Rn); \\
\text{integer } m & = \text{UInt}(Rm); \\
\end{array}
\]

\[
\begin{align*}
\text{if } & \text{size} == '01' || \text{size} == '10' \text{ then UNDEFINED;} \\
\text{if } & \text{size} == '11' && \text{!HaveBit128PMULLExt()} \text{ then UNDEFINED;} \\
\text{integer } & \text{esize} = 8 \ll \text{UInt}(\text{size}); \\
\text{integer } & \text{datasize} = 64; \\
\text{integer } & \text{part} = \text{UInt}(Q); \\
\text{integer } & \text{elements} = \text{datasize} \div \text{esize};
\end{align*}
\]

\(<Vd>\)

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

\(<Ta>\)

Is an arrangement specifier, encoded in “size”:

\[
\begin{array}{c|c}
\text{size} & \text{<Ta>} \\
\hline
00 & 8\text{H} \\
01 & \text{RESERVED} \\
10 & \text{RESERVED} \\
11 & '1Q' \\
\end{array}
\]

The ‘1Q’ arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

\(<Vn>\)

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

\(<Tb>\)

Is an arrangement specifier, encoded in “size:Q”:

\[
\begin{array}{c|c|c}
\text{size} & \text{Q} & \text{<Tb>} \\
\hline
00 & 0 & 8\text{B} \\
00 & 1 & 16\text{B} \\
01 & \text{x} & \text{RESERVED} \\
10 & \text{x} & \text{RESERVED} \\
11 & 0 & 1\text{D} \\
11 & 1 & 2\text{D} \\
\end{array}
\]

\(<Vm>\)

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
bite(element1);
bite(element2);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about memory accesses, see Load/Store addressing modes.

### Assembler Symbols

| <prfop> | Is the prefetch operation, defined as <type><target><policy>.<br><br><type> is one of:<br>PLD | Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.<br>PLI | Preload instructions, encoded in the "Rt<4:3>" field as 0b01.<br>PST | Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.<br><br><target> is one of:<br>L1 | Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.<br>L2 | Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.<br>L3 | Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.<br><br><policy> is one of:<br>KEEP | Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.<br>STRM | Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.<br>For more information on these prefetch operations, see Prefetch memory.<br>For other encodings of the "Rt" field, use <imm5>.<br>Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.<br>This syntax is only for encodings that are not accessible using <prfop>.
<br><imm5> | 
<br><Xn|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rt" field.<br><pimm> | Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.
integer n = \texttt{UInt}(Rn);
integer t = \texttt{UInt}(Rt);
\texttt{AccType} acctype = \texttt{AccType\_NORMAL};
\texttt{MemOp} memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then \texttt{MemOp\_LOAD} else \texttt{MemOp\_STORE};
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = \texttt{MemOp\_PREFETCH};
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = \texttt{MemOp\_LOAD};
    if size == '10' \&\& opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 \texttt{<!scale>};
if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE; // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if ! postindex then
  address = address + offset;

  case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
      data = Mem[address, datasize DIV 8, acctype];
      if signed then
        X[t] = SignExtend(data, regsize);
      else
        X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
      Prefetch(address, t<4:0>);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  else if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about memory accesses, see Load/Store addressing modes.

LITERAL

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & \text{imm19} & \text{Rt} \\
\end{array}
\]

opc

integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);

Assembler Symbols

\(<\text{prfop}>\)\|<\text{imm5}>\), <\text{label}>

Is the prefetch operation, defined as \(<\text{type}>\)<\text{target}>\,<\text{policy}>.
\(<\text{type}>\) is one of:

**PLD**

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

**PLI**

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

**PST**

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

\(<\text{target}>\) is one of:

**L1**

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

**L2**

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

**L3**

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

\(<\text{policy}>\) is one of:

**KEEP**

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

**STRM**

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.
For more information on these prefetch operations, see Prefetch memory.
For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.
This syntax is only for encodings that are not accessible using <prfop>.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

**Operation**

```plaintext
bits(64) address = PC[] + offset;
bits(size*8) data;

case memop of
    when MemOp LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;
    when MemOp PREFETCH
        Prefetch(address, t<4:0>);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about memory accesses, see Load/Store addressing modes.

Integer

PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>)], <extend> {<amount>})

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;             // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.
    <type> is one of:
        PLD
            Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
        PLI
            Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
        PST
            Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.
    <target> is one of:
        L1
            Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
        L2
            Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
        L3
            Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.
    <policy> is one of:
        KEEP
            Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
        STRM
            Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.
    This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted.
<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>

### Shared Decode

```plaintext
type integer n = UInt(Rn);
type integer t = UInt(Rt);
type integer m = UInt(Rm);
type AccType acctype = AccType_NORMAL;
type MemOp memop;
type boolean signed;
type integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
    integer datasize = 8 << scale;
```

```
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;}
PRFM (unscaled offset)

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about memory accesses, see Load/Store addressing modes.

For information about memory accesses, see Load/Store addressing modes.

### Unscaled offset

PRFUM (<prfop>|#<imm5>), [<Xn|SP>(, #<simm>)]

- boolean wback = FALSE;
- boolean postindex = FALSE;
- integer scale = UInt(size);
- bits(64) offset = SignExtend(imm9, 64);

### Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

- <type> is one of:
  - PLD: Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
  - PLI: Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
  - PST: Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

- <target> is one of:
  - L1: Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
  - L2: Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
  - L3: Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

- <policy> is one of:
  - KEEP: Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
  - STRM: Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory.

For other encodings of the "Rt" field, use <imm5>.

- <imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rt" field.

- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  end_case;

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLPSL);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  end_case;

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];
if ! postindex then
  address = address + offset;
end_case

  case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;
      end_if
    when MemOp_LOAD
      data = Mem[address, datasize DIV 8, acctype];
      if signed then
        X[t] = SignExtend(data, regsize);
      else
        X[t] = ZeroExtend(data, regsize);
  end_case

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a NOP.

System
(ARMv8.2)

```
<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

System

```
SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111'  
    SEE "XPACLRI";
  when '0001 xxx'  
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction();                  // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction();     // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction();         // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSB;
  when '0010 101'
    op = SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);           
  otherwise EndOfInstruction();                                 // Instruction executes as NOP
```

SEE "XPACLRI";
case op of
when SystemHintOp_YIELD
    Hint_Yield();
when SystemHintOp_WFE
    if IsEventRegisterSet() then
        ClearEventRegister();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch64.CheckForWFXTrap(EL1, TRUE);
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFXTrap(EL2, TRUE);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFXTrap(EL3, TRUE);
            WaitForEvent();
    when SystemHintOp_WFI
        if !InterruptPending() then
            if PSTATE.EL == EL0 then
                // Check for traps described by the OS which may be EL1 or EL2.
                AArch64.CheckForWFXTrap(EL1, FALSE);
            if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
                // Check for traps described by the Hypervisor.
                AArch64.CheckForWFXTrap(EL2, FALSE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                AArch64.CheckForWFXTrap(EL3, FALSE);
                WaitForInterrupt();
    when SystemHintOp_SEV
        SendEvent();
    when SystemHintOp_SEVL
        SendEventLocal();
    when SystemHintOp_ESB
        SynchronizeErrors();
        AArch64.ESBOperation();
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
            AArch64.vESBOperation();
            TakeUnmaskedSErrorInterrupts();
    when SystemHintOp_PSB
        ProfilingSynchronizationBarrier();
    when SystemHintOp_TSB
        TraceSynchronizationBarrier();
    when SystemHintOp_CSDB
        ConsumptionOfSpeculativeDataBarrier();
    when SystemHintOp_BTI
        BTypeNext = '00';

otherwise // do nothing
Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

```plaintext
<table>
<thead>
<tr>
<th>CRm&lt;3:0&gt;</th>
<th>CRm&lt;1:0&gt;</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1</td>
<td>0 1 0 1</td>
<td>0 0</td>
</tr>
</tbody>
</table>
```

**System**

```plaintext
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if HaveSBExt() && CRm<3:0> == '0000' then
      op = MemBarrierOp_SB;
    else
      UNDEFINED;

case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
  otherwise
    if CRm<3:2> == '01' then
      op = MemBarrierOp_PSSBB;
    elsif CRm<3:2> == '00' && opc == '00' then
      op = MemBarrierOp_SSBB;
    elsif HaveSBExt() && CRm<3:2> == '00' && opc == '11' then
      op = MemBarrierOp_SB;
    else
      types = MBReqTypes_All;
      domain = MBReqDomain_FullSystem;
```

PSSBB
case op of
    when MemBarrierOp_DSB
        DataSynchronizationBarrier(domain, types);
    when MemBarrierOp_DMB
        DataMemoryBarrier(domain, types);
    when MemBarrierOp_ISB
        InstructionSynchronizationBarrier();
    when MemBarrierOp_SSBB
        SpeculativeSynchronizationBarrierToVA();
    when MemBarrierOp_PSSBB
        SpeculativeSynchronizationBarrierToPA();
    when MemBarrierOp_SB
        SpeculationBarrier();
RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see ADDHN.

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| {2} | U | Q | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 1 | 0 | 0 | 0 | Rn | Rd | o1 |

Three registers, not all the same type

RADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');

Assembler Symbols

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>Ta</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```plaintext
CheckFPAdvSIMEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;
for e = 0 to elements-1
  element1 = Elem[operand1, e, 2*esize];
  element2 = Elem[operand2, e, 2*esize];
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  sum = sum + round_const;
  Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RAX1

Rotate and Exclusive OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

Advanced SIMD
(ARMv8.2)

<table>
<thead>
<tr>
<th></th>
<th>SIMD&amp;FP Destination Register</th>
<th>SIMD&amp;FP Source Register 1</th>
<th>SIMD&amp;FP Source Register 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>1 1 0 0 1 1 0 0 1 1</td>
<td>1 0 0 0 1 1</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Advanced SIMD

\[
\text{RAX1} \quad \langle V_d \rangle.2D, \; \langle V_n \rangle.2D, \; \langle V_m \rangle.2D
\]

\[
\text{if} \; \neg \text{HaveSHA3Ext}() \; \text{then UNDEFINED;}
\]

\[
\begin{align*}
\text{integer} \; d &= \text{UInt}(\text{Rd}); \\
\text{integer} \; n &= \text{UInt}(\text{Rn}); \\
\text{integer} \; m &= \text{UInt}(\text{Rm});
\end{align*}
\]

Assembler Symbols

- \langle V_d \rangle is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \langle V_n \rangle is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \langle V_m \rangle is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

\[
\text{if} \; \neg \text{HaveSHA3Ext}() \; \text{then UNDEFINED;}
\]

\[
\begin{align*}
d &= \text{UInt}(\text{Rd}); \\
n &= \text{UInt}(\text{Rn}); \\
m &= \text{UInt}(\text{Rm}); \\
\end{align*}
\]

\[
\begin{align*}
\text{bits}(128) \; V_m &= V[m]; \\
\text{bits}(128) \; V_n &= V[n]; \\
\quad V[d] &= V_n \text{ EOR} \left( \text{ROL}(V_m<127:64>, 1); \text{ROL}(V_m<63:0>, 1) \right).
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
RBIT (vector)

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 0 0 1 1 0 0 0 0 0 0 1 0 1 1 0 Rd
Rn
```

**Vector**

```
RBIT <Vd>.<T>, <Vn>.<T>
```

- integer d = `UInt` (Rd);
- integer n = `UInt` (Rn);
- integer esize = 8;
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV 8;

**Assembler Symbols**

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in “Q”:
  - Q  <T>
    - 0  8B
    - 1  16B
- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
bits(esize) rev;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  for i = 0 to esize-1
    rev<esize-1-i> = element<i>;
  Elem[result, e, esize] = rev;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RBIT

Reverse Bits reverses the bit order in a register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

32-bit (sf == 0)

RBIT <Wd>, <Wn>

64-bit (sf == 1)

RBIT <Xd>, <Xn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 64;
```

Assembler Symbols

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;
for i = 0 to datasize-1
    result<datasize-1-i> = operand<i>;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

### Integer

```python
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));
if !pac && m != 0 then UNDEFINED;
elsif pac && !HavePACExt() then UNDEFINED;
case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;
if pac then
    if Z == '0' && m != 31 then UNDEFINED;
    if branch_type == BranchType_RET then
        if n != 31 then UNDEFINED;
        n = 30;
        source_is_sp = TRUE;
```

### Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.
bits(64) target = X[n];
case branch_type of
  when BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
  if pac then
    bits(64) modifier = if source_is_sp then SP[] else X[m];
    if use_key_a then
      target = AuthIA(target, modifier);
    else
      target = AuthIB(target, modifier);
  if branch_type == BranchType_INDCALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
RETURN FROM SUBROUTINE, WITH POINTER AUTHENTICATION. THIS INSTRUCTION AUTHENTICATES THE ADDRESS THAT IS HELD IN LR, USING SP AS THE MODIFIER AND THE SPECIFIED KEY, BRANCHES TO THE AUTHENTICATED ADDRESS, WITH A HINT THAT THIS INSTRUCTION IS A SUBROUTINE RETURN.

KEY A IS USED FOR RETAA, AND KEY B IS USED FOR RETAB.

IF THE AUTHENTICATION Passes, THE PE CONTINUES EXECUTION AT THE TARGET OF THE BRANCH. IF THE AUTHENTICATION FAILS, A Translation fault IS GENERATED. THE AUTHENTICATED ADDRESS IS NOT WRITTEN BACK TO LR.

```
integer 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
   1  1  0  1  0  1  1  0  1  0  1  1  1  1  0  0  0  0  1  M  1  1  1  1  1  1  1  1
   Z op A Rn Rm
```

RETAA (M == 0)

RETAA

RETAB (M == 1)

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));
if !pac && m != 0 then
    UNDEFINED;
elsif pac && !HavePACExt() then
    UNDEFINED;
case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;
if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;
    if branch_type == BranchType_RET then
        if n != 31 then UNDEFINED;
        n = 30;
        source_is_sp = TRUE;
```
Operation

\[
\begin{align*}
\text{bits(64) target} &= X[n] ; \\
\text{case branch_type of} & \\
& \qquad \text{when BranchType INDIRECT} \\
& \quad \quad \text{if InGuardedPage then} \\
& \quad \quad \quad \text{if } n == 16 || n == 17 \text{ then} \\
& \quad \quad \quad \quad \text{BTypeNext} = '01'; \\
& \quad \quad \quad \text{else} \quad \text{BTypeNext} = '11'; \\
& \quad \quad \quad \text{else} \quad \text{BTypeNext} = '01'; \\
& \quad \quad \text{when BranchType INDCALL} \\
& \quad \quad \quad \text{BTypeNext} = '10'; \\
& \quad \quad \text{when BranchType RET} \\
& \quad \quad \quad \text{BTypeNext} = '00'; \\
& \text{if pac then} \\
& \quad \text{bits(64) modifier} = \text{if source_is_sp then SP[]} \text{ else X}[m]; \\
& \quad \text{if use_key_a then} \\
& \quad \quad \text{target} = \text{AuthIA}(\text{target}, \text{modifier}); \\
& \quad \text{else} \\
& \quad \quad \text{target} = \text{AuthIB}(\text{target}, \text{modifier}); \\
& \text{if branch_type == BranchType INDCALL then} \ X[30] = PC[] + 4; \\
& \text{BranchTo(target, branch_type);}
\end{align*}
\]
Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REVD.

```
| sf | 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| opc|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
```

32-bit (sf == 0 && opc == 10)

```
REV <Wd>, <Wn>
```

64-bit (sf == 1 && opc == 11)

```
REV <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;
```

**Assembler Symbols**

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

```
b[datasize] operand = [X][n];
b[datasize] result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index + 7:rev_index> = operand<index + 7:index>;
    index = index + 8;
    rev_index = rev_index - 8;
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
REV16 (vector)

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

integer d = Uint(Rd);
integer n = Uint(Rn);

// size=size:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << Uint(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//    8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if Uint(op)+Uint(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

| sf | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |

32-bit (sf == 0)

REV16 <<Wd>>, <<Wn>>

64-bit (sf == 1)

REV16 <<Xd>>, <<Xn>>

integer d = Uint(Rd);
integer n = Uint(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00' unreachable();
  when '01' container_size = 16;
  when '10' container_size = 32;
  when '11' if sf == '0' then UNDEFINED;
                   container_size = 64;

Assembler Symbols

<<Wd>> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<<Wn>> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<<Xd>> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<<Xn>> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index + 7:rev_index> = operand<index + 7:index>;
    index = index + 8;
    rev_index = rev_index - 8;
X[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
REV32 (vector)

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| U | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Rd |

Vector

REV32 <Vd>.<T>, <Vn>.<T>

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=size: B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op)+UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
when '10' container_size = 16;
when '01' container_size = 32;
when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
\end{verbatim}

Assembler Symbols

\begin{verbatim}
<Vd>     Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>     Is an arrangement specifier, encoded in "size-Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>lx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>     Is the name of the SIMD&FP source register, encoded in the "Rn" field.
\end{verbatim}
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn</td>
<td></td>
</tr>
</tbody>
</table>

64-bit

REV32 <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00' Unreachable();
  when '01' container_size = 16;
  when '10' container_size = 32;
  when '11' if sf == '0' then UNDEFINED;
                container_size = 64;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index + 7:rev_index> = operand<index + 7:index>;
    index = index + 8;
    rev_index = rev_index - 8;

X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

integer \( d = \) UInt(Rd);
integer \( n = \) UInt(Rn);

// size=esize: B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op)+UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;

Assembler Symbols

\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{T}>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<\text{Vn}>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**REV64**

Reverse Bytes reverses the byte order in a 64-bit general-purpose register. When assembling for ARMv8.2, an assembler must support this pseudo-instruction. It is **OPTIONAL** whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than ARMv8.2.

This is a pseudo-instruction of REV. This means:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode for this instruction.

### 64-bit

```
REV64 <Xd>, <Xn>
```

is equivalent to

```
REV <Xd>, <Xn>
```

### Assembler Symbols

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

### Operation

The description of REV gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RMIF

Performs a rotation right of a value held in a general purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer

(ARMv8.4)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

sf

Integer

RMIF <Xn>, #<shift>, #<mask>

if !HaveFlagManipulateExt() || sf != '1' then UNDEFINED;
integer lsb = UInt(imm6);
integer n = UInt(Rn);

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift> Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
<mask> Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

Operation

bits(4) tmp;
bits(64) tmpreg = X[n];
tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
if mask<3> == '1' then PSTATE.N = tmp<3>;
if mask<2> == '1' then PSTATE.Z = tmp<2>;
if mask<1> == '1' then PSTATE.C = tmp<1>;
if mask<0> == '1' then PSTATE.V = tmp<0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of EXTR. This means:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>N</th>
<th>0</th>
<th>Rm</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

32-bit (sf == 0 && N == 0 && imms == 0xxxxx)

ROR <Wd>, <Ws>, #<shift>

is equivalent to

EXTR <Wd>, <Ws>, <Ws>, #<shift>

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1 && N == 1)

ROR <Xd>, <Xs>, #<shift>

is equivalent to

EXTR <Xd>, <Xs>, <Xs>, #<shift>

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Ws> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xs> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<shift> For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field.
For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of RORV. This means:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode for this instruction.

32-bit (sf == 0)

ROR <Wd>, <Wn>, <Wm>

is equivalent to

RORV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

ROR <Xd>, <Xn>, <Xm>

is equivalent to

RORV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RORV

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register).

<table>
<thead>
<tr>
<th>sf</th>
<th>Rm</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

RORV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

RORV <Xd>, <Xn>, <Xm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```plaintext
bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
  - The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
RSHRN, RSHRN2

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SHRN.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

RSHRN[2] <Vd>,<Tb>, <Vn>,<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”: 
### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
for e = 0 to elements-1
    element = (UINT(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RSUBHN, RSUBHN2

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see SUBHN.

The RSUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | size | 1  | Rm | 0  | 1  | 1  | 0  | 0  | Rn | Rd |
| U  | o1 |

Three registers, not all the same type

RSUBHN[2] <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;
for e = 0 to elements-1
  element1 = Elem[operand1, e, 2*esize];
  element2 = Elem[operand2, e, 2*esize];
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  sum = sum + round_const;
  Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
\end{verbatim}
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register, while the SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm | 0  | 1  | 0  | 1  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |
|    | U  | op |

Three registers, not all the same type

SABAL[2] <Vd>,<Ta>, <Vn>,<Tb>, <Vm>,<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size-Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | Q  | 0  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm  | 0  | 1  | 1  | 1  | 0  | 1  | Rn  | 0  | 1  | Rd  | ac |

Three registers of the same type

SABD `<Vd>`., `<T>`, `<Vn>`., `<T>`, `<Vm>`./.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

`<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` is an arrangement specifier, encoded in “size-Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

`<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)esize-1:0;  
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABDL instruction writes the vector to the lower half of the destination register and clears the upper half, while the SABDL2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three registers, not all the same type

\texttt{SABDL(2)} \texttt{<Vd>.<Ta>, <Vn>.<Tb>}, \texttt{<Vm>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');

Assembler Symbols

\begin{center}
\begin{tabular}{|c|c|}
\hline
Q & 2  \\
\hline
0 & [absent] \\
1 & [present] \\
\hline
\end{tabular}
\end{center}

\texttt{<Vd>} \quad Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

\texttt{<Ta>} \quad Is an arrangement specifier, encoded in “size”:

\begin{center}
\begin{tabular}{|c|c|}
\hline
size & \texttt{<Ta>} \\
\hline
00 & 8H \\
01 & 4S \\
10 & 2D \\
11 & RESERVED \\
\hline
\end{tabular}
\end{center}

\texttt{<Vn>} \quad Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\texttt{<Tb>} \quad Is an arrangement specifier, encoded in “size:Q”:

\begin{center}
\begin{tabular}{|c|c|}
\hline
size & Q & \texttt{<Tb>} \\
\hline
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & RESERVED \\
\hline
\end{tabular}
\end{center}

\texttt{<Vm>} \quad Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bites(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U Q 0 0 1 1 1 0 size 1 0 0 0 0 0 1 1 0 1 0 Rd

Vector

SADALP <Vd>.<Ta>, <Vn>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) sum;
integer op1;
integer op2;
result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register, while the SADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0   1   0   1   1   0   size 1   Rm  0 0 0 0 0 0   Rn           Rd |
| U   o1               |
```

**Three registers, not all the same type**

SADDL(2) \(<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Ta>\) Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;Ta&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\) Is an arrangement specifier, encoded in “size-Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>(&lt;Tb&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SADDLP

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places
the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the
source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an
attempt to execute the instruction might be trapped.

Vector

SADDLP <Vd>.<Ta>, <Vn>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
\end{verbatim}

\textbf{Assembler Symbols}

\texttt{<V>} Is the destination width specifier, encoded in “\textit{size}”:

\begin{center}
\begin{tabular}{|c|c|}
\hline
size & <V> \\
\hline
00 & H \\
01 & S \\
10 & D \\
11 & RESERVED \\
\hline
\end{tabular}
\end{center}

\texttt{<d>} Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

\texttt{<Vn>} Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\texttt{<T>} Is an arrangement specifier, encoded in “\textit{size:Q}”:

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
size & Q & <T> \\
\hline
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & RESERVED \\
10 & 1 & 4S \\
11 & x & RESERVED \\
\hline
\end{tabular}
\end{center}

\textbf{Operation}

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);
V[d] = sum2*esize-1:0;
\end{verbatim}
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register. The SADDW instruction extracts the second source vector from the lower half of the second source register, while the SADDW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 1  | Rm | 0  | 0  | 0  | 1  | 0  | Rn | Rd | o1 |

**Three registers, not all the same type**

SADDW(2) \(<Vd>\), \(<Ta>\), \(<Vn>\), \(<Vm>\), \(<Tb>\)

<table>
<thead>
<tr>
<th>(Q)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Ta>\) Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;Ta&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

\(<Tb>\) Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>(Q)</th>
<th>(&lt;Tb&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Speculation Barrier is a barrier that controls speculation. The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

### System

```
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if HaveSBExt() && CRm<3:0> == '0000' then
      op = MemBarrierOp_SB;
    else
      UNDEFINED;
  end case;

case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
  otherwise
    if CRm<3:2> == '01' then
      op = MemBarrierOp_PSSBB;
    elsif CRm<3:2> == '00' && opc == '00' then
      op = MemBarrierOp_SSBB;
    elsif HaveSBExt() && CRm<3:2> == '00' && opc == '11' then
      op = MemBarrierOp_SB;
    else
      types = MBReqTypes_All;
      domain = MBReqDomain_FullSystem;
  end case;
```
case op of
    when MemBarrierOp_DSB
        DataSynchronizationBarrier(domain, types);
    when MemBarrierOp_DMB
        DataMemoryBarrier(domain, types);
    when MemBarrierOp_ISB
        InstructionSynchronizationBarrier();
    when MemBarrierOp_SSBB
        SpeculativeSynchronizationBarrierToVA();
    when MemBarrierOp_PSSBB
        SpeculativeSynchronizationBarrierToPA();
    when MemBarrierOp_SB
        SpeculationBarrier();
SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC.

32-bit (sf == 0)

SBC <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SBC <Xd>, <Xn>, <Xm>

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGC</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGC</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;

if sub_op then
    operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS.

32-bit (sf == 0)

```
SBCS <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
SBCS <Xd>, <Xn>, <Xm>
```

Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGCS</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;

if sub_op then
    operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SBFIZ

Signed Bitfield Insert in Zeros copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 0  | 0  | 1  | 0  | 1  | 1  | 0  | N  | immr| imms| Rn  | Rd  |

**32-bit (sf == 0 & N == 0)**

SBFIZ <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

SBFM <Wd>, <Wn>, #(<lsb> MOD 32), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

**64-bit (sf == 1 & N == 1)**

SBFIZ <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

SBFM <Xd>, <Xn>, #(<lsb> MOD 64), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

**Assembler Symbols**

- <Wd>: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>: Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd>: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>: Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <lsb>: For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- <width>: For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

**Operation**

The description of SBFM gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If `<imms>` is greater than or equal to `<immr>`, this copies a bitfield of `<imms>-<immr>+1` bits starting from bit position `<immr>` in the source register to the least significant bits of the destination register.

If `<imms>` is less than `<immr>`, this copies a bitfield of `<imms>+1` bits from the least significant bits of the source register to bit position `(regsize-<immr>)` of the destination register, where `regsize` is the destination register size of 32 or 64 bits.

In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases **ASR (immediate)**, **SBFIZ**, **SBFX**, **SXTB**, **SXTH**, and **SXTW**.

### 32-bit (sf == 0 && N == 0)

```
SBFM <Wd>, <Wn>, #<immr>, #<imms>
```

### 64-bit (sf == 1 && N == 1)

```
SBFM <Xd>, <Xn>, #<immr>, #<imms>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

case opc of
    when '00' inzero = TRUE; extend = TRUE; // SBFM
    when '01' inzero = FALSE; extend = FALSE; // BFM
    when '10' inzero = TRUE; extend = FALSE; // UBFM
    when '11' UNDEFINED;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;
R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<immr>` For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.
  For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
- `<imms>` For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.
  For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.
## Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR (immediate)</td>
<td>32-bit</td>
<td>imms == '011111'</td>
</tr>
<tr>
<td>ASR (immediate)</td>
<td>64-bit</td>
<td>imms == '111111'</td>
</tr>
<tr>
<td>SBFIZ</td>
<td></td>
<td>unint(imms) &lt; unint(immr)</td>
</tr>
<tr>
<td>SBFX</td>
<td></td>
<td>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</td>
</tr>
<tr>
<td>SXTB</td>
<td></td>
<td>immr == '000000' &amp;&amp; imms == '000111'</td>
</tr>
<tr>
<td>SXTH</td>
<td></td>
<td>immr == '000000' &amp;&amp; imms == '001111'</td>
</tr>
<tr>
<td>SXTW</td>
<td></td>
<td>immr == '000000' &amp;&amp; imms == '011111'</td>
</tr>
</tbody>
</table>

## Operation

```plaintext
bits(datasize) dst = if inzero then Zeros() else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SBFX

Signed Bitfield Extract copies a bitfield of \(<width>\) bits starting from bit position \(<lsb>\) in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of \(SBFM\). This means:

- The encodings in this description are named to match the encodings of \(SBFM\).
- The description of \(SBFM\) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccccccccc}
\hline
sf & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & N & \hline
\text{immr} & \text{imms} & \text{Rn} & \text{Rd} \\
\end{array}
\]

\text{opc}

**32-bit (sf == 0 && N == 0)**

SBFX \(<Wd>, <Wn>, \#<lsb>, \#<width>\)

is equivalent to

SBFM \(<Wd>, <Wn>, \#<lsb>, \{(<lsb>) + (<width> - 1)}\)

and is the preferred disassembly when \(BFXPreferred(sf, opc<1>, imms, immr)\).

**64-bit (sf == 1 && N == 1)**

SBFX \(<Xd>, <Xn>, \#<lsb>, \#<width>\)

is equivalent to

SBFM \(<Xd>, <Xn>, \#<lsb>, \{(<lsb>) + (<width> - 1)}\)

and is the preferred disassembly when \(BFXPreferred(sf, opc<1>, imms, immr)\).

**Assembler Symbols**

- \(<Wd>\): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\): Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\): Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<lsb>\): For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
  - For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- \(<width>\): For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  - For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

**Operation**

The description of \(SBFM\) gives the operational pseudocode for this instruction.

**Operational Information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SCVTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 1  | 1  | 0  | 0  | 1  | Rd |
| U  | immh |

#### SCVTF <V><d>, <V><n>, #<fbits>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == 'lxxx' then 64 else if immh == '0lxx' then 32 else 16;
integer datasize = esize;
integer elements = 1;
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Vector

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 1  | 1  | 0  | 0  | 1  | Rd |
| U  | immh |

#### SCVTF <Vd>, <T>, #<fbits>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == 'lxxx' then 64 else if immh == '0lxx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

- `<V>`: Is a width specifier, encoded in “immh”:
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>\texttt{SEE Advanced SIMD modified immediate}</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
<td>\texttt{RESERVED}</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>\texttt{RESERVED}</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits>
For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>\texttt{RESERVED}</td>
</tr>
<tr>
<td>001x</td>
<td>(32-$\text{Uint}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\texttt{SEE Advanced SIMD modified immediate}</td>
</tr>
<tr>
<td>0001</td>
<td>\texttt{RESERVED}</td>
</tr>
<tr>
<td>001x</td>
<td>(32-$\text{Uint}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();

\text{bits}(\text{datasize}) \ \text{operand} = V[n];
\text{bits}(\text{datasize}) \ \text{result};
\text{bits}(\text{esize}) \ \text{element};

\text{for } e = 0 \text{ to elements-1}
\quad \text{element} = \text{Elem}[\text{operand}, e, \text{esize}];
\quad \text{Elem}[\text{result}, e, \text{esize}] = \text{FixedToFP}(\text{element}, \text{fracbits}, \text{unsigned}, \text{FPCR}, \text{rounding});
\quad V[d] = \text{result};
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision.

### Scalar half precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | Rn | Rd |
|-----------------|-----------------|
| 0 1 0 1 1 1 1 0 0 1 1 1 0 0 1 1 1 0 1 1 0 |       |     |
| U               |

#### Scalar half precision

SCVTF \(<Hd>, <Hn>\)

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

### Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | Rn | Rd |
|-----------------|-----------------|
| 0 1 0 1 1 1 1 0 0 0 0 0 1 1 1 0 1 1 0 |       |     |
| U               |

#### Scalar single-precision and double-precision

SCVTF \(<V><d>, <V><n>\)

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

### Vector half precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | Rn | Rd |
|-----------------|-----------------|
| 0 Q 0 0 1 1 1 0 0 1 1 1 0 0 1 1 1 0 1 1 0 |       |     |
| U               |
Vector half precision

```
SCVTF <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

```
Vector single-precision and double-precision

SCVTF <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Hd&gt;</td>
<td>Is the 16-bit name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Hn&gt;</td>
<td>Is the 16-bit name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;V&gt;</td>
<td>Is a width specifier, encoded in &quot;sz&quot;:</td>
</tr>
<tr>
<td></td>
<td>sz  &lt;V&gt;</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>&lt;d&gt;</td>
<td>Is the number of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;n&gt;</td>
<td>Is the number of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;Vd&gt;</td>
<td>Is the name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;T&gt;</td>
<td>For the vector half precision variant: is an arrangement specifier, encoded in “Q”:</td>
</tr>
<tr>
<td></td>
<td>Q  &lt;T&gt;</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:</td>
</tr>
<tr>
<td></td>
<td>sz Q &lt;T&gt;</td>
</tr>
<tr>
<td></td>
<td>0 0</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
</tr>
<tr>
<td>&lt;Vn&gt;</td>
<td>Is the name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);
V[d] = result;
```
SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the $FPCR$, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in $FPCR$, the exception results in either a flag being set in $FPSR$, or a synchronous exception being generated. For more information, see $Floating-point exception traps$.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 1 0</th>
<th>type</th>
<th>0 0 0 1 0</th>
<th>scale</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>
```

The instruction uses the following fields:

- **sf**: Signed flag
- **type**: Instruction type (0 for SCVTF)
- **scale**: Scale value
- **Rn**: Source register
- **Rd**: Destination register

The instruction sets the rounding mode according to the $FPCR$. If a floating-point exception is generated, it is handled according to the settings in $FPSR$ or $FPCR$. The instruction is also subject to the Security and Exception level restrictions defined by the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers.
32-bit to half-precision (sf == 0 && type == 11)
(ARMv8.2)

    SCVTF <Hd>, <Wn>, #<fbits>

32-bit to single-precision (sf == 0 && type == 00)

    SCVTF <Sd>, <Wn>, #<fbits>

32-bit to double-precision (sf == 0 && type == 01)

    SCVTF <Dd>, <Wn>, #<fbits>

64-bit to half-precision (sf == 1 && type == 11)
(ARMv8.2)

    SCVTF <Hd>, <Xn>, #<fbits>

64-bit to single-precision (sf == 1 && type == 00)

    SCVTF <Sd>, <Xn>, #<fbits>

64-bit to double-precision (sf == 1 && type == 01)

    SCVTF <Dd>, <Xn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    fi
fi

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
    when '00 11'        // FCVTZ
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    otherwise
        UNDEFINED;
fi

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".

For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
    V[d] = fltval;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>type</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmode</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
32-bit to half-precision (sf == 0 && type == 11) (ARMv8.2)

SCVTF <Hd>, <Wn>

32-bit to single-precision (sf == 0 && type == 00)

SCVTF <Sd>, <Wn>

32-bit to double-precision (sf == 0 && type == 01)

SCVTF <Dd>, <Wn>

64-bit to half-precision (sf == 1 && type == 11) (ARMv8.2)

SCVTF <Hd>, <Xn>

64-bit to single-precision (sf == 1 && type == 00)

SCVTF <Sd>, <Xn>

64-bit to double-precision (sf == 1 && type == 01)

SCVTF <Dd>, <Xn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  case opcode<2:1>:rmode of
    when '00 xx'        // FCVT[NPMZ][US]
      rounding = FPDecodeRounding(rmode);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '01 00'        // [US]CVTF
      rounding = FPRoundingMode(FPCR);
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_ItoF;
    when '10 00'        // FCVTA[US]
      rounding = FPRounding_TIEAWAY;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI;
    when '11 00'        // FMOV
      if fltsize != 16 && fltsize != intsize then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
      part = 0;
    when '11 01'        // FMOV D[1]
      if intsize != 64 || fltsize != 128 then UNDEFINED;
      op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
      part = 1;
      fltsize = 64; // size of D[1] is 64
    when '11 11'        // FJCVTZS
      if !HaveFJCVTZSExt() then UNDEFINED;
      rounding = FPRounding_ZERO;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.

### 32-bit (sf == 0)

```
SDIV <Wd>, <Wn>, <Wm>
```

### 64-bit (sf == 1)

```
SDIV <Xd>, <Xn>, < Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean unsigned = (o1 == '0');
```

### Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `< Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

### Operation

```
bits(datasize) operand1 = X[n];
b bits(datasize) operand2 = X[m];
integer result;
if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, unsigned)) / Real(Int(operand2, unsigned)));
X[d] = result<datasize-1:0>;
```
SDOT (by element)

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it. 

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector
(ARMv8.2)

$$\text{SDOT } <Vd>, <Ta>, <Vn>, <Tb>, <Vm>.4B[<index>]$$

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U=='0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 < UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Ta>$ Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>Ta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Tb>$ Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>Tb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

$<Vm>$ Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

$<\text{index}>$ Is the element index, encoded in the "H:L" fields.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```
SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Three registers of the same type

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | size| 0  | Rm | 1  | 0  | 1  | 0  | 1  | Rn | 1  | 0  | 1  | 0  | 1  | Rn | 1  | 0  | 1  | 0  | Rd |

Three registers of the same type

SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveDOTPExt() then UNDEFINED;
if size!='10' then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
checkFPAdvSIMEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
SETF8, SETF16

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. SETF8 treats the value as an 8 bit value, and SETF16 treats the value as an 16 bit value.

The PSTATE.C flag is not affected by these instructions.

Integer

(ARMv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | sz | 0  | 0  | 1  | 0  | Rn | 0  | 1  | 1  | 0  | 1  |    |      |

sf

SETF8 (sz == 0)

SETF8 <Wn>

SETF16 (sz == 1)

SETF16 <Wn>

if !HaveFlagManipulateExt() || sf != '0' then UNDEFINED;
integer msb = if sz=='1' then 15 else 7;
integer n = UInt(Rn);

Assembler Symbols

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

bits(32) tmprreg = X[n];
PSTATE.N = tmprreg<msb>;
PSTATE.Z = if (tmprreg<msb:0> == Zeros(msb+1)) then '1' else '0';
PSTATE.V = tmprreg<msb+1> EOR tmprreg<msb>;
//PSTATE.C unchanged;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see **Wait for Event mechanism and Send event**.

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 1 0 0 1 1 1 1
```

System

```
SEV

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111' SEE "XPACLRI";
  when '0001 xxx' SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0011 xxx' SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0' op = SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction(); // Instruction executes as NOP
```

SEV
case op of
  when SystemHintOp_YIELD
      Hint_Yield();
  when SystemHintOp_WFE
      if IsEventRegisterSet() then
          ClearEventRegister();
      else
          if PSTATE.EL == EL0 then
              // Check for traps described by the OS which may be EL1 or EL2.
              AArch64.CheckForWFxTrap(EL1, TRUE);
          if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
              // Check for traps described by the Hypervisor.
              AArch64.CheckForWFxTrap(EL2, TRUE);
          if HaveEL(EL3) && PSTATE.EL != EL3 then
              // Check for traps described by the Secure Monitor.
              AArch64.CheckForWFxTrap(EL3, TRUE);
          WaitForEvent();
  when SystemHintOp_WFI
      if !InterruptPending() then
          if PSTATE.EL == EL0 then
              // Check for traps described by the OS which may be EL1 or EL2.
              AArch64.CheckForWFxTrap(EL1, FALSE);
          if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
              // Check for traps described by the Hypervisor.
              AArch64.CheckForWFxTrap(EL2, FALSE);
          if HaveEL(EL3) && PSTATE.EL != EL3 then
              // Check for traps described by the Secure Monitor.
              AArch64.CheckForWFxTrap(EL3, FALSE);
          WaitForInterrupt();
  when SystemHintOp_SEV
      SendEvent();
  when SystemHintOp_SEVL
      SendEventLocal();
  when SystemHintOp_ESB
      SynchronizeErrors();
      AArch64.ESBOperation();
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
      TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
      ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
      TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
      ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
      BTypeNext = '00';
  otherwise // do nothing
Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**CRm op2**

**System**

**SEVL**

```c
SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111'
    SEE "XPACLRI";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction();               // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction();  // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction();       // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIAZP, PACIBZ, PACIBZP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBZP";
  when '0100 xx0'
    op = SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction();                               // Instruction executes as NOP
```

SEVL
case op of
    when SystemHintOp_YIELD
        Hint_Yield();
    when SystemHintOp_WFE
        if IsEventRegisterSet() then
            ClearEventRegister();
        else
            if PSTATE.EL == EL0 then
                // Check for traps described by the OS which may be EL1 or EL2.
                AArch64.CheckForWFxTrap(EL1, TRUE);
            if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
                // Check for traps described by the Hypervisor.
                AArch64.CheckForWFxTrap(EL2, TRUE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                AArch64.CheckForWFxTrap(EL3, TRUE);
        WaitForEvent();
    when SystemHintOp_WFI
        if !InterruptPending() then
            if PSTATE.EL == EL0 then
                // Check for traps described by the OS which may be EL1 or EL2.
                AArch64.CheckForWFxTrap(EL1, FALSE);
            if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
                // Check for traps described by the Hypervisor.
                AArch64.CheckForWFxTrap(EL2, FALSE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                AArch64.CheckForWFxTrap(EL3, FALSE);
        WaitForInterrupt();
    when SystemHintOp_SEV
        SendEvent();
    when SystemHintOp_SEVL
        SendEventLocal();
when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
when SystemHintOp_TSB
    TraceSynchronizationBarrier();
when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
when SystemHintOp_BTI
    BTypeNext = '00';
otherwise // do nothing
SHA1 hash update (choose).

```
0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0
Rm
0 0 0 0 0 0 0 0 0
Rn
0 0 0 0 0 0 0 0 0
Rd
```

Advanced SIMD

```
SHA1C <Qd>, <Sn>, <Vm>.4S
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

- `<Qd>`: Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- `<Sn>`: Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32)  Y = V[n];  // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32)  t;
for e = 0 to 3
    t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y : X, 32);
V[d] = X;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1H

SHA1 fixed rotate.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rn  Rd

Advanced SIMD

SHA1H <Sd>, <Sn>

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(32) operand = V[n];  // read element [0] only, [1-3] zeroed
V[d] = ROL(operand, 30);

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SHA1M

SHA1 hash update (majority).

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Rm | 0 | 0 | 1 | 0 | 0 | 0 | Rn | Rd |

Advanced SIMD

SHA1M \(<Qd>, <Sn>, <Vm>\).

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rm);
if \(!\text{HaveSHA1Ext}()\) then UNDEFINED;

Assembler Symbols

\(<Qd>\) Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
\(<Sn>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
\(<Vm>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

\text{AArch64.CheckFPAdvSIMDEnabled}();

\[
\begin{align*}
\text{bits}(128) \ X &= V[d]; \\
\text{bits}(32) \ Y &= V[n]; & \quad \text{Note: 32 not 128 bits wide} \\
\text{bits}(128) \ W &= V[m]; \\
\text{bits}(32) \ t; \\
\text{for } e = 0 \text{ to } 3 \\
\quad t &= \text{SHAmajority}(X<63:32>, X<95:64>, X<127:96>); \\
\quad Y &= Y + \text{ROL}(X<31:0>, 5) + t + \text{Elem}[W, e, 32]; \\
\quad X<63:32> &= \text{ROL}(X<63:32>, 30); \\
\quad <Y, X> &= \text{ROL}(Y : X, 32); \\
\quad V[d] &= X;
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1P

SHA1 hash update (parity).

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Rm| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Rn| 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Rd| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Advanced SIMD

SHA1P \(<Qd>, <Sn>, <Vm>\).4S

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

\(<Qd>\) Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.

\(<Sn>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```plaintext
AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32)  Y = V[n];     // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32)  t;
for e = 0 to 3
  t = SHAparity(X\(63:32\), X\(95:64\), X\(127:96\));
  Y = Y + ROL(X\(31:0\), 5) + t + Elem(W, e, 32);
  X\(63:32\) = ROL(X\(63:32\), 30);
  <Y, X> = ROL(Y : X, 32);
V[d] = X;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1SU0

SHA1 schedule update 0.

```
0 1 0 1 1 1 0 0 0 0
Rm 0 0 1 1 0 0
Rn
Rd
```

Advanced SIMD

SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
result = operand2<63:0> : operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1SU1

SHA1 schedule update 1.

Advanced SIMD

SHA1SU1 <Vd>.4S, <Vn>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0> = ROL(T<31:0>, 1);
result<63:32> = ROL(T<63:32>, 1);
result<95:64> = ROL(T<95:64>, 1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA256H2

SHA256 hash update (part 2).

Advanced SIMD

SHA256H2 <Qd>, <Qn>, <Vm>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
boolean part1 = (P == '0');

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
if part1 then
    result = SHA256hash(V[d], V[n], V[m], TRUE);
else
    result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SHA256H

SHA256 hash update (part 1).

Advanced SIMD

SHA256H <Qd>, <Qn>, <Vm>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
boolean part1 = (P == '0');

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
if part1 then
    result = SHA256hash(V[d], V[n], V[m], TRUE);
else
    result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Advanced SIMD

SHA256SU0 <Vd>.4S, <Vn>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand2<31:0> : operand1<127:32>;
bits(32) elt;
for e = 0 to 3
  elt = Elem[T, e, 32];
  elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
  Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA256SU1

SHA256 schedule update 1.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Advanced SIMD

SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

\[
\text{AArch64.CheckFPAdvSIMDEnabled}();
\]

\[
\begin{align*}
\text{bits}(128) \text{ operand1} &= V[d]; \\
\text{bits}(128) \text{ operand2} &= V[n]; \\
\text{bits}(128) \text{ operand3} &= V[m]; \\
\text{bits}(128) \text{ result} &= \text{operand3<31:0> : operand2<127:32>}; \\
\text{bits}(64) \text{ T1} &= \text{operand3<127:64>}; \\
\text{bits}(32) \text{ elt} &= \text{T1} ; \\
\text{for e = 0 to 1} \quad \text{elt} &= \text{Elem}[T1, e, 32]; \\
& \quad \text{elt} &= \text{ROR}(\text{elt}, 17) \text{ EOR ROR}(\text{elt}, 19) \text{ EOR LSR}(\text{elt}, 10); \\
& \quad \text{elt} &= \text{elt} + \text{Elem}[\text{operand1}, e, 32] + \text{Elem}[T0, e, 32]; \\
& \quad \text{Elem}[\text{result}, e, 32] = \text{elt}; \\
\text{T1} &= \text{result<63:0>}; \\
\text{for e = 2 to 3} \quad \text{elt} &= \text{Elem}[T1, e - 2, 32]; \\
& \quad \text{elt} &= \text{ROR}(\text{elt}, 17) \text{ EOR ROR}(\text{elt}, 19) \text{ EOR LSR}(\text{elt}, 10); \\
& \quad \text{elt} &= \text{elt} + \text{Elem}[\text{operand1}, e, 32] + \text{Elem}[T0, e, 32]; \\
& \quad \text{Elem}[\text{result}, e, 32] = \text{elt}; \\
V[d] &= \text{result};
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

Advanced SIMD

(ARMv8.2)

```
1 1 0 0 1 1 0 0 1 1 | Rd 1 0 0 0 0 1 | Rn | Rd
```

Advanced SIMD

SHA512H2 <Qd>, <Qn>, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) NSigma0;
bits(64) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];
NSigma0 = ROR(Y<63:0>, 28) EOR ROR(Y<63:0>,34) EOR ROR(Y<63:0>,39);
Vtmp<127:64> = (X<63:0> AND Y<127:64>) EOR (X<63:0> AND Y<63:0>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + W<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>,34) EOR ROR(Vtmp<127:64>,39);
Vtmp<63:0> = (Vtmp<127:64> AND Y<63:0>) EOR (Vtmp<127:64> AND Y<127:64>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + NSigma0 + W<63:0>);
V[d] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register. This instruction is implemented only when ARMv8.2-SHA is implemented.

Advanced SIMD
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  1 |  1 |  0 |  1 |  1 |  1 |  0 |  1 |  1 |  0 |  0 |  1 |  1 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |

Advanced SIMD

SHA512H \(<Q_d>, <Q_n>, <V_m>.2D\)

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

\(<Q_d>\) Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
\(<Q_n>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
\(<V_m>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) MSigma1;
bits(64) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

MSigma1 = ROR(Y<127:64>, 14) EOR ROR(Y<127:64>,18) EOR ROR(Y<127:64>,41);
Vtmp<127:64> = (Y<127:64> AND X<63:0>) EOR (NOT(Y<127:64>) AND X<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + W<127:64>);
tmp = Vtmp<127:64> + Y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp,18) EOR ROR(tmp,41);
Vtmp<63:0> = (tmp AND Y<127:64>) EOR (NOT(tmp) AND X<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + W<63:0>);
v[d] = Vtmp;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

Advanced SIMD
(ARMv8.2)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Rn   Rd

Advanced SIMD

SHA512SU0 \(<V_d>.2D, \ <V_n>.2D\)

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

Assembler Symbols

\(<V_d>\) Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

\(<V_n>\) Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig0;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) W = V[d];
sig0 = ROR(W<127:64>, 1) EOR ROR(W<127:64>, 8) EOR '0000000':W<127:71>);
Vtmp<63:0> = W<63:0> + sig0;
sig0 = ROR(X<63:0>, 1) EOR ROR(X<63:0>, 8) EOR '0000000':X<63:7>);
Vtmp<127:64> = W<127:64> + sig0;
V[d] = Vtmp;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA512SU1

SHA512 Schedule Update 1 takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

Advanced SIMD (ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|-----------------|
| 1 1 0 0 1 1 1 0 0 1 1 | Rm | 1 0 0 0 1 0 | Rn | Rd |

Advanced SIMD

SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig1;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

sig1 = ROR(X<127:64>, 19) EOR ROR(X<63:0>, 19) EOR ('000000':X<127:70>);
Vtmp<127:64> = W<127:64> + sig1 + Y<127:64>;
sig1 = ROR(X<63:0>, 19) EOR ROR(X<63:0>, 61) EOR ('000000':X<63:6>);
Vtmp<63:0> = W<63:0> + sig1 + Y<63:0>;
V[d] = Vtmp;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see *SRHADD*.

Depending on the settings in the *CPACR_EL1, CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Integer \( d \) = \( \text{UInt}(Rd) \);

Integer \( n \) = \( \text{UInt}(Rn) \);

Integer \( m \) = \( \text{UInt}(Rm) \);

If \( \text{size} == '11' \) then UNDEFINED;

Integer \( \text{esize} = 8 << \text{UInt}(\text{size}) \);

Integer \( \text{datasize} = \) if \( Q == '1' \) then 128 else 64;

Integer \( \text{elements} = \text{datasize} \div \text{esize} \);

Boolean \( \text{unsigned} = (U == '1') \);

**Assembler Symbols**

- \(<Vd>\) is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<T>\) is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- \(<Vn>\) is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \(<Vm>\) is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SHL

Shift Left (immediate). This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 1   | 1   | 1   | 1   | 1   | 0   | ! = 0000 | immb | 0   | 0   | 1   | 0   | 1   | Rn  | Rd  |

**immh**

```plaintext
Scalar

SHL <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = UInt(immh:immb) - esize;
```

### Vector

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 0   | 1   | 1   | 1   | 1   | 0   | ! = 0000 | immb | 0   | 0   | 1   | 0   | 1   | Rn  | Rd  |

**immh**

```plaintext
Vector

SHL <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = UInt(immh:immb) - esize;
```

### Assembler Symbols

- `<V>` is a width specifier, encoded in “immh”:
  
  | 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
  |-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
  | 0xxx | RESERVED |
  | lxxx  | D        |

- `<d>` is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb) - 64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb) - 8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb) - 16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb) - 32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb) - 64)</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
for e = 0 to elements-1
    Elem[result, e, esize] = LSL(Elem[operand, e, esize], shift);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SHLL, SHLL2**

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The **SHLL** instruction extracts vector elements from the lower half of the source register, while the **SHLL2** instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Binary Representation](image)

**Vector**

\[
\text{SHLL}([2]) \cdot <Vd>, <Ta>, <Vn>, <Tb>, \#<shift>
\]

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);

if \( \text{size} == '11' \) then UNDEFINED;
integer \( \text{esize} = 8 << \text{UInt}(\text{size}) \);
integer \( \text{datasize} = 64 \);
integer \( \text{part} = \text{UInt}(Q) \);
integer \( \text{elements} = \text{datasize} \div \text{esize} \);
integer \( \text{shift} = \text{esize} \);
integer \( \text{unsigned} = \text{FALSE}; // \text{Or TRUE without change of functionality} \)

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Ta>\) Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\) Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<\text{shift}>\) Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SHRN, SHRN2**

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see [RSHRN](#).

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector

```
SHRN(2) <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `if immh == '0000' then SEE(asimdimm);`
- `if immh<3> == '1' then UNDEFINED;`
- `integer esize = 8 << HighestSetBit(immh);`
- `integer datasize = 64;`
- `integer part = UInt(Q);`
- `integer elements = datasize DIV esize;`
- `integer shift = (2 * esize) - UInt(immh:immb);`
- `boolean round = (op == '1');`

### Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>[absent]</td>
<td>[present]</td>
</tr>
</tbody>
</table>

#### <Vd>
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

- `immh Q <Tb>`: Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

#### <Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

#### <Ta>
Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

#### <shift>
Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”: 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 | Q | 0 | 1 | 1 | 1 | 1 | 0 | != 0000 | immh | 1 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | oper |

---

*SHRN, SHRN2 Page 1069*
<table>
<thead>
<tr>
<th>immh</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
  element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
  Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHSUB

Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

```
SHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Assembler Symbols

```
<integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
for e = 0 to elements-1
   element1 = Int(Elem[operand1, e, esize], unsigned);
   element2 = Int(Elem[operand2, e, esize], unsigned);
   diff = element1 - element2;
   Elem[result, e, esize] = diff<esize:1>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SLI

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

The following figure shows an example of the operation of shift left by 3 for an 8-bit vector element.

![Figure showing the operation of SLI]

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| immh | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn  | Rd |

Scalar

SLI <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

Vector

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| immh | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn  | Rd |

Vector

SLI <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
Assembler Symbols

<i>V</i> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;i&gt;V&lt;/i&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<i>d</i> Is the number of the SIMD&FP destination register, in the "Rd" field.

<i>n</i> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<i>Vd</i> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<i>T</i> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;i&gt;T&lt;/i&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<i>Vn</i> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<i>shift</i> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;i&gt;shift&lt;/i&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immh) - 64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;i&gt;shift&lt;/i&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immh) - 8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immh) - 16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immh) - 32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immh) - 64)</td>
</tr>
</tbody>
</table>

Operation

```plaintext
CheckFPAvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shifted;
for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when ARMv8.2-SM is implemented.

Advanced SIMD

(ARMv8.2)

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |  Rm |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |  Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |  Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>,15):ROL(Vm<95:64>,15):ROL(Vm<63:32>,15));
for i = 0 to 3
  if i == 3 then
    result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>,15));
  result<(32*i)+31:(32*i)> = result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>,15) EOR ROL(result<(32*i)+31:(32*i)>,23);
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when **ARMv8.2-SM** is implemented.

### Advanced SIMD

**(ARMv8.2)**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Rd| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  |
| Rn| 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  |
| Rm| 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  |

### Advanced SIMD

SM3PARTW2 \(<Vd>.4S, <Vn>.4S, <Vm>.4S\)

```plaintext
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

### Assembler Symbols

- \(<Vd>\) Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- \(<Vn>\) Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- \(<Vm>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```plaintext
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
bits(128) tmp;
bits(32) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>,7);ROL(Vm<95:64>,7);ROL(Vm<63:32>,7);ROL(Vm<31:0>,7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>,15);
tmp2 = tmp2 EOR ROL(tmp2,15) EOR ROL(tmp2,23);
result<127:96> = result<127:96> EOR tmp2;
V[d]= result;
```

### Operational information

If **PSTATE.DIT** is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3SS1

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other
32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7
and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being
written to 0.

This instruction is implemented only when ARMv8.2-SM is implemented.

Advanced SIMD
(ARMv8.2)

Advanced SIMD

SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the ”Rd” field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the ”Rn” field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the ”Rm” field.
<Va> Is the name of the third SIMD&FP source register, encoded in the ”Ra” field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) Va = V[a];
Vd<127:96> = ROL((ROL(Vn<127:96>,12) + Vm<127:96> + Va<127:96>) , 7);
Vd<95:0> = Zeros();
V[d] = Vd;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when ARMv8.2-SM is implemented.

Advanced SIMD

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 1  | 0  | imm2 | 0  | 0  | Rn | Rd |

Advanced SIMD

SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```plaintext
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```plaintext
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;
WjPrime = Elem[Vm,i,32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>,12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1 + Vd<31:0> + SS2 + WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>,9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;
```

SM3TT1A
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3TT1B takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when ARMv8.2-SM is implemented.

Advanced SIMD

(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | imm2| 0  | 1  | Rd |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**SM3TT1B**

SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S<imm2>

**Assembler Symbols**

- `<Vd>` is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- `<Vn>` is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- `<imm2>` is a 32-bit element indexed out of `<Vm>`, encoded in "imm2".

**Operation**

```assembly
AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;
WjPrime = Elem[Vm,i,32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>,12);
TT1 = (Vd<127:96> AND Vd<63:32>) OR (Vd<127:96> AND Vd<95:64>) OR (Vd<63:32> AND Vd<95:64>);
TT1 = (TT1 + Vd<31:0> + SS2 + WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>,9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3TT2A

SM3TT2A takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when ARMv8.2-SM is implemented.

Advanced SIMD
(ARMv8.2)

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm2</td>
<td>1</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Advanced SIMD

SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S<imm2>

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT1;

Wj = Elem[Vm,i,32];
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2 + Vd<31:0> + Vn<127:96> + Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>,19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2,9) EOR ROL(TT2,17);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when ARMv8.2-SM is implemented.

Advanced SIMD
(ARMv8.2)

<table>
<thead>
<tr>
<th>Rm</th>
<th>imm2</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>


if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm,i,32];
TT2 = (Vd<127:96> AND Vd<95:64>) OR (NOT(Vd<127:96>) AND Vd<63:32>);
TT2 = (TT2 + Vd<31:0> + Vn<127:96> + Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>,19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2,9) EOR ROL(TT2,17);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM4E

SM4 Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SM is implemented.

Advanced SIMD

(ARMv8.2)

Advanced SIMD

SM4E <Vd>.4S, <Vn>.4S

if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vn = V[n];
bits(32) intval;
bits(8) sboxout;
bits(128) roundresult;
bits(32) roundkey;
roundresult = V[d];
for index = 0 to 3
    roundkey = Elem[Vn,index,32];
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;
    for i = 0 to 3
        Elem[intval,i,8] = Sbox(Elem[intval,i,8]);
        intval = intval EOR ROL(intval,2) EOR ROL(intval,10) EOR ROL(intval,18) EOR ROL(intval,24);
    roundresult<31:0> = roundresult<63:32> EOR roundresult<95:64> EOR roundresult<127:96> EOR roundresult<127:96> = intval;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
- The values of the NZCV flags.
SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SM is implemented.

Advanced SIMD
(ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  |

Advanced SIMD

SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S

if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(32) intval;
bits(8) sboxout;
bits(128) result;
bits(32) const;
bits(128) roundresult;

roundresult = V[n];
for index = 0 to 3
    const = Elem[Vm,index,32];
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
    for i = 0 to 3
        Elem[intval,i,8] = Sbox(Elem[intval,i,8]);
        intval = intval EOR ROl(intval,13) EOR ROl(intval,23);
    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
V[d] = roundresult;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit

SMADDL \(<X_d>, <W_n>, <W_m>, <X_a>\)

integer \(d = \text{UInt}(R_d)\);
integer \(n = \text{UInt}(R_n)\);
integer \(m = \text{UInt}(R_m)\);
integer \(a = \text{UInt}(R_a)\);
integer destsize = 64;
integer datasure = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

\(<X_d>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<W_n>\) Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

\(<W_m>\) Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

\(<X_a>\) Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMULL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

\([X_d] = result<63:0>\);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMAX

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

SMAX <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

\[
\text{SMAXP } <\text{Vd}.<\text{T}>, <\text{Vn}.<\text{T}>, <\text{Vm}.<\text{T}> \]

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
integer \( m = \text{UInt}(Rm) \);
if size == '11' then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{T}>\) Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<\text{Vn}>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Vm}>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = \( V[n] \);
bits(datasize) operand2 = \( V[m] \);
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
   element1 = Int(Elem[concat, 2*e, esize], unsigned);
   element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
   maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
   Elem[result, e, esize] = maxmin<esize-1:0>;
\( V[d] = result \);
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMAXV

Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| U  | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| Rn | Rd |
```

Advanced SIMD

SMAXV <$V<$d>, <$Vn>$<$T>

```
integer d = Uint(Rd);
integer n = Uint(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << Uint(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

```
<V> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;
maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d] = maxmin<esize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMC

Secure Monitor Call causes an exception to EL3. SMC is available only for software executing at EL1 or higher. It is undefined in EL0.

If the values of \texttt{HCR\_EL2.TSC} and \texttt{SCR\_EL3.SMD} are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in \texttt{ESR\_ELx}, using the EC value 0x17, that is taken to EL3.

If the value of \texttt{HCR\_EL2.TSC} is 1, execution of an SMC instruction in a Non-secure EL1 state generates an exception that is taken to EL2, regardless of the value of \texttt{SCR\_EL3.SMD}. For more information, see \textit{Traps to EL2 of Non-secure EL1 execution of SMC instructions}.

If the value of \texttt{HCR\_EL2.TSC} is 0 and the value of \texttt{SCR\_EL3.SMD} is 1, the SMC instruction is undefined.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  |

\texttt{bits(16) imm = imm16;}

Assembler Symbols

\texttt{<imm>} Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

\begin{verbatim}
AArch64.CheckForSMCUndefOrTrap(imm);
if SCR_EL3.SMD == '1' then
  // SMC disabled
  AArch64.UndefinedFault();
else
  AArch64.CallSecureMonitor(imm);
\end{verbatim}

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00ber8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMIN

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembly Code

```assembly
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Operation

```assembly
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMINP

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

\texttt{SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Assembler Symbols

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<T>} Is an arrangement specifier, encoded in “size-Q”:

- \texttt{integer d} = \texttt{UInt}(Rd);
- \texttt{integer n} = \texttt{UInt}(Rn);
- \texttt{integer m} = \texttt{UInt}(Rm);
- if size == '11' then UNDEFINED;
- \texttt{integer esize} = 8 << \texttt{UInt}(size);
- \texttt{integer datasize} = if Q == '1' then 128 else 64;
- \texttt{integer elements} = datasize DIV esize;
- \texttt{boolean unsigned} = (U == '1');
- \texttt{boolean minimum} = (o1 == '1');

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
  element1 = Int(Elem[concat, 2*e, esize], unsigned);
  element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
\end{verbatim}
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```

Advanced SIMD

SMINV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
  element = Int(Elem[operand, e, esize], unsigned);
  maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMLAL, SMLAL2 (by element)

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The SMLAL instruction extracts vector elements from the lower half of the first source register, while the SMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U Q 0 0 1 1 1 1 size L M Rm 0 0 1 0 H 0 Rn Rd 0 2
```

Vector

\[
\text{SMLAL}[2] \langle Vd\rangle .\langle Ta\rangle , \langle Vn\rangle .\langle Tb\rangle , \langle Vm\rangle .\langle Ts\rangle [\langle index\rangle]
\]

```plaintext
integer idxxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

```
  Q  2
0 [absent]
1 [present]
```

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

\(<Ta>\) Is an arrangement specifier, encoded in “size”:

```
size  <Ta>
00  RESERVED
01  4S
10  2D
11  RESERVED
```

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\) Is an arrangement specifier, encoded in “size:Q”:
<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(idxsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLAL instruction extracts each source vector from the lower half of each source register, while the SMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>1</th>
<th>Rm</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three registers, not all the same type

SMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMLSL, SMLSL2 (by element)

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts vector elements from the lower half of the first source register, while the SMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | Q | 0 | 1 | 1 | 1 | size | L | M | Rm | 0 | 1 | 1 | 0 | H | 0 | Rn | Rd |
   |   |   |   |   |   |      |   |   |    |   |   |   |   |   |   |   |   |   |
   | U |   |   |   |   |   |      |   |   |    |   |   |   |   |   |   |   |   |   |
```

Vector

SMLSL[2] <Vd>..<Ta>, <Vn>..<Tb>, <Vm>..<Ts>[<index>]

```plaintext
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datataset = 64;
integer part = UInt(Q);
integer elements = datataset DIV esize;
boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

```
Q  2
 0 | [absent] |
 1 | [present] |
```

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

```
size <Ta>
 00 | RESERVED 
 01 | 4S     
 10 | 2D     
 11 | RESERVED 
```

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

SMLSL, SMLSL2 (by element)
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vm>**

Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

**<Ts>**

Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<index>**

Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts each source vector from the lower half of each source register, while the SMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | Q  | 0  | 1  | 1  | 0  | size | 1  | Rm | 1  | 0  | 1  | 0  | 0  | Rn  | Rd  |
| U  | o1 |    |    |    |    |      |    |    |    |    |    |    |    |      |    |

Three registers, not all the same type

SMLSL[2] <Vd>,<Ta>, <Vn>,<Tb>, <Vm><Tb>

```plaintext
typelet d = UInt(Rd);
typelet n = UInt(Rn);
typelet m = UInt(Rm);

if size == '11' then UNDEFINED;
typelet esize = 8 << UInt(size);
typelet datasize = 64;
typelet part = UInt(Q);
typelet elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

- <Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.
- <Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- <Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.
- <Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- <Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = Y[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
Y[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMNEGL

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of SMSUBL. This means:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode for this instruction.

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  1  0  0  1  1  0  1  1  0  0  1  Rm  1  1  1  1  1  Rn  Rd
```

- U 00 Ra

**64-bit**

SMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

SMSUBL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

**Assembler Symbols**

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

The description of SMSUBL gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZC flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZC flags.
SMOV

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the $\text{CPACR\_EL1}$, $\text{CPTR\_EL2}$, and $\text{CPTR\_EL3}$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (Q == 0)

$\text{SMOV} \ <Wd>, \ <Vn>\.\langle Ts\rangle[\langle index\rangle]$  

64-reg,SMOV-64-reg (Q == 1)

$\text{SMOV} \ <Xd>, \ <Vn>\.\langle Ts\rangle[\langle index\rangle]$  

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
  when 'xxxxx1' size = 0;     // SMOV [WX]d, Vn.B
  when 'xxxx10' size = 1;     // SMOV [WX]d, Vn.H
  when '1xx100' size = 2;     // SMOV Xd, Vn.S
  otherwise     UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

Assembler Symbols

$\langle Wd\rangle$ Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

$\langle Xd\rangle$ Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

$\langle Vn\rangle$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.

$\langle Ts\rangle$ For the 32-bit variant: is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
</tbody>
</table>

For the 64-reg,SMOV-64-reg variant: is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xxx10</td>
<td>S</td>
</tr>
</tbody>
</table>

$\langle index\rangle$ For the 32-bit variant: is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
</tbody>
</table>

For the 64-reg,SMOV-64-reg variant: is the element index encoded in “imm5”:
# Operation

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
X[d] = SignExtend(Elem[operand, index, esize], datasize);
```

## Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to
the 64-bit destination register.

This instruction is used by the alias **SMNEGL**.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  |

### Assembler Symbols

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- **<Xa>** Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SMNEGL</strong></td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer destsize = 64;
integer datasiize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer destsize = 64;
integer datasiize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

if sub_op then
    result = Int(operand3, unsigned) - (Int(operand1, unsigned) * Int(operand2, unsigned));
else
    result = Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SMULH**

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

```
0 0 1 1 0 1 0
  1 0
U  Rm
  0 (1) (1) (1) (1) Rn
  Ra Rd
```

64-bit

**SMULH <Xd>, <Xn>, <Xm>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra); // ignored by UMULH/SMULH
integer destsize = 64;
integer datasize = destsize;
boolean unsigned = (U == '1');

**Assembler Symbols**

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

integer result;
result = Int(operand1, unsigned) * Int(operand2, unsigned);
X[d] = result<127:64>;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register, while the SMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

2  
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  
Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm>  
Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>M</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Restricted to V0-V15 when element size <Ts> is H.

Is an element size specifier, encoded in “size”:

```
size <Ts>
00 RESERVED
01 H
10 S
11 RESERVED
```

Is the element index, encoded in “size:L:H:M”:

```
size <index>
00 RESERVED
01 H:L:M
10 H:L
11 RESERVED
```

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = \[m\];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SMULL, SMULL2 (vector)**

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts each source vector from the lower half of each source register, while the SMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Register Format](image)

Three registers, not all the same type

```
SMULL{2} <Vd>..<Ta>., <Vn>..<Tb>., <Vm>..<Tb>
```

Integer d = UInt(Rd);
Integer n = UInt(Rn);
Integer m = UInt(Rm);

if size == '11' then UNDEFINED;
Integer esize = 8 << UInt(size);
Integer datasize = 64;
Integer part = UInt(Q);
Integer elements = datasize DIV esize;

Boolean unsigned = (U == '1');

**Assembler Symbols**

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64()

bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1 * element2)<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of SMADDL. This means:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1 0 0 1 1 0 1 1 0 0 1 Rm 0 1 1 1 1 1 Rn Rd

U o0 Ra

64-bit

SMULL <Xd>, <Wn>, <Wm>

is equivalent to

SMADDL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SQABS

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

Scalar

```
SQABS <V>d>, <V>n>
```

```python
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

Vector

```
SQABS <Vd>.<T>, <Vn>.<T>
```

```python
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

- `<V>` is a width specifier, encoded in “size”:
  ```
  | size | <V>
  |----------------------------------|
  | 00   | B
  | 01   | H
  | 10   | S
  | 11   | D
  ```

- `<d>` is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>` is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in “size:Q”:
<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQADD**

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit $FPSCR.QC$ is set.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

![Scalar Encoding](image)

 Scalar $\langle V\rangle\langle d\rangle$, $\langle V\rangle\langle n\rangle$, $\langle V\rangle\langle m\rangle$

integer $d$ = $\text{UInt}(Rd)$;  
integer $n$ = $\text{UInt}(Rn)$;  
integer $m$ = $\text{UInt}(Rm)$;  
integer $esize$ = 8 $<<\text{UInt}(size)$;  
integer $datasize$ = $esize$;  
integer $elements$ = 1;  
boolean $unsigned$ = ($U$ == '1');

### Vector

![Vector Encoding](image)

 Vector $\langle Vd\rangle$.T$, $\langle Vn\rangle$.T$, $\langle Vm\rangle$.T$

integer $d$ = $\text{UInt}(Rd)$;  
integer $n$ = $\text{UInt}(Rn)$;  
integer $m$ = $\text{UInt}(Rm)$;  
if $size:Q$ == '110' then UNDEFINED;  
integer $esize$ = 8 $<<\text{UInt}(size)$;  
integer $datasize$ = if $Q$ == '1' then 128 else 64;  
integer $elements$ = $datasize$ DIV $esize$;  
boolean $unsigned$ = ($U$ == '1');

**Assembler Symbols**

$\langle V\rangle$ Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>$\langle V\rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

$\langle d\rangle$ Is the number of the SIMD&FP destination register, in the "Rd" field.

$\langle n\rangle$ Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

$\langle m\rangle$ Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

$\langle Vd\rangle$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMLAL, SQDMLAL2 (by element)**

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit `FPSR.QC` is set.

The `SQDMLAL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>size</th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>H</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>

**Scalar**

`SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]`

- integer idxdsiz = if H == '1' then 128 else 64;
- integer index;
- bit Rmhi;
- case size of
  - when '01' index = `UInt(H:L:M)`, Rmhi = '0';
  - when '10' index = `UInt(H:L)`, Rmhi = M;
  - otherwise UNDEFINED;

- integer d = `UInt(Rd)``;
- integer n = `UInt(Rn)``;
- integer m = `UInt(Rmhi:Rm)``;

- integer esize = 8 << `UInt(size)`;
- integer datasize = esize;
- integer elements = 1;
- integer part = 0;
- boolean sub_op = (o2 == '1');

### Vector

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>size</th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>H</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>
Vector

SQDMLAL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = Uint(H:L:M); Rmhi = '0';
  when '10' index = Uint(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rmhi:Rm);

integer esize = 8 << Uint(size);
integer datasize = 64;
integer part = Uint(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

Assembler Symbols

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>
Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Va>
Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>
Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vb>
Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64() {
  bits(datasize) operand1 = Vpart[n, part];
  bits(idxdsize) operand2 = V[m];
  bits(2*datasize) operand3 = V[d];
  bits(2*datasize) result;
  integer element1;
  integer element2;
  integer product;
  integer accum;
  boolean sat1;
  boolean sat2;
  element2 = SInt(Elem[operand2, index, esize]);
  for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
      accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
      accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
      (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';
  V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit $FPSR.QC$ is set.

The SQDMLAL instruction extracts each source vector from the lower half of each source register, while the SQDMLAL2 instruction extracts each source vector from the upper half of each source register. Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
0 1 0 1 1 1 1 0 | size 1 | Rm 1 0 0 1 0 0 | Rn | Rd
```

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

**Vector**

```
0 0 0 1 1 1 1 0 | size 1 | Rm 1 0 0 1 0 0 | Rn | Rd
```

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = datasize DIV esize;
integer part = UInt(Q);

boolean sub_op = (o1 == '1');
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;

integer element1;
integer element2;

bits(2*datasize) product;

integer accum;

boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMLSL, SQDMLSL2 (by element)**

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit $FPSR.QC$ is set.

The SQDMLSL instruction extracts vector elements from the lower half of the first source register, while the SQDMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
   0  1  0  1  1  1  1  size  L  M  Rm  0  1  1  1  H  0  Rn  Rd  o2
```

**Scalar**

```
SQDMLSL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = $UInt(H:L:M); Rmhi = '0';
  when '10' index = $UInt(H:L);   Rmhi = M;
  otherwise UNDEFINED;

integer d = $UInt(Rd);
integer n = $UInt(Rn);
integer m = $UInt(Rmhi:Rm);

integer esize = 8 << $UInt(size);
integer datasize = esize;
integer elements = l;
integer part = 0;
boolean sub_op = (o2 == '1');

**Vector**

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
   0  Q  0  1  1  1  1  size  L  M  Rm  0  1  1  1  H  0  Rn  Rd  o2
```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

Assembler Symbols

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding
the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size &lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Va> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size &lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<Vb> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size &lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

```
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
intrinsic element2;
integer element3;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';
V[d] = result;
```
**SQDMLSL, SQDMLSL2 (vector)**

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit $FPSR.QC$ is set.

The $SQDMLSL$ instruction extracts each source vector from the lower half of each source register, while the $SQDMLSL2$ instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | size | 1  | Rm | 1  | 0  | 1  | 1  | 0  | 0  | Rn | 0  | Rd |
| o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Scalar**

$SQDMLSL\ <Va>d, <Vb>n, <Vb>m$

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm | 1  | 0  | 1  | 1  | 0  | 0  | Rn | 0  | Rd |
| o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Vector**

$SQDMLSL(2)\ <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>$

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q".
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q &lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bites(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem(operand1, e, esize));
    element2 = SInt(Elem(operand2, e, esize));
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem(operand3, e, 2*esize)) - SInt(product);
    else
        accum = SInt(Elem(operand3, e, 2*esize)) + SInt(product);
    (Elem(result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';
```

V[d] = result;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMULH (by element)**

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see **SQRDMULH**.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

$$\text{SQDMULH } <V><d>, <V><n>, <Vm>..<Ts>[<index>]$$

```plaintext
integer idxdsz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean round = (op == '1');
```

**Vector**

$$\text{SQDMULH (by element)}$$
Vector

\texttt{SQDMULH} \langle V_d \rangle, \langle V_n \rangle, \langle T \rangle, \langle V_m \rangle, \langle V_s \rangle [\langle \text{index} \rangle]

\begin{verbatim}
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
\textbf{case size of}
  when '01' index = \texttt{UInt}(H:L:M); Rmhi = '0';
  when '10' index = \texttt{UInt}(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rmhi:Rm);

integer esize = 8 << \texttt{UInt}(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
\end{verbatim}

**Assembler Symbols**

\texttt{<V>} Is a width specifier, encoded in “size”:

\begin{tabular}{|c|c|}
\hline
\texttt{size} & <V> \\
\hline
00 & RESERVED \\
01 & H \\
10 & S \\
11 & RESERVED \\
\hline
\end{tabular}

\texttt{<d>} Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<n>} Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\texttt{<V_d>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<T>} Is an arrangement specifier, encoded in “size:Q”:

\begin{tabular}{|c|c|}
\hline
\texttt{size} & <T> \\
\hline
00 & x RESERVED \\
01 & 0 4H \\
01 & 1 8H \\
10 & 0 2S \\
10 & 1 4S \\
11 & x RESERVED \\
\hline
\end{tabular}

\texttt{<V_n>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\texttt{<V_m>} Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

\begin{tabular}{|c|c|}
\hline
\texttt{size} & <V_m> \\
\hline
00 & RESERVED \\
01 & 0:Rm \\
10 & M:Rm \\
11 & RESERVED \\
\hline
\end{tabular}

Restricted to V0-V15 when element size \texttt{<Ts>} is H.

\texttt{<Ts>} Is an element size specifier, encoded in “size”:

\begin{tabular}{|c|c|}
\hline
\texttt{size} & <Ts> \\
\hline
00 & RESERVED \\
01 & H \\
10 & S \\
11 & RESERVED \\
\hline
\end{tabular}

\texttt{<index>} Is the element index, encoded in “size:L:H:M”:

SQDMULH (by element)
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  product = (2 * element1 * element2) + round_const;
  // The following only saturates if element1 and element2 equal -(2^(esize-1))
  (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
  if sat then FPSR.QC = '1';
V[d] = result;
```
**SQDMULH (vector)**

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see **SQRDMULH**.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm  | 1  | 0  | 1  | 1  | 0  | 1  | Rd  | |
|    |    |    |    |    |    |    | U    |    |     |    |    |    |    |    |    |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**SQDMULH** `<V><d>, <V><n>, <V><m>`

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `integer m = UInt(Rm);`
- `if size == '11' || size == '00' then UNDEFINED;`
- `integer esize = 8 << UInt(size);`
- `integer datasize = esize;`
- `integer elements = 1;`
- `boolean rounding = (U == '1');`

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm  | 1  | 0  | 1  | 1  | 0  | 1  | Rd  | |
|    |    |    |    |    |    |    |    | U    |    |     |    |    |    |    |    |    |    |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**SQDMULH** `<Vd>, <T>, <Vn>, <T>, <Vm>, <T>`

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `integer m = UInt(Rm);`
- `if size == '11' || size == '00' then UNDEFINED;`
- `integer esize = 8 << UInt(size);`
- `integer datasize = if Q == '1' then 128 else 64;`
- `integer elements = datasize DIV esize;`
- `boolean rounding = (U == '1');`

### Assembler Symbols

- `<V>`
  - Is a width specifier, encoded in "size":
    
    | size | `<V>` |
    |------|-------|
    | 00   | RESERVED |
    | 01   | H      |
    | 10   | S      |
    | 11   | RESERVED |

- `<d>`
  - Is the number of the SIMD&FP destination register, in the "Rd" field.

- `<n>`
  - Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \texttt{FPSR}.QC is set.

The \texttt{SQDMULL} instruction extracts the first source vector from the lower half of the first source register, while the \texttt{SQDMULL2} instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the \texttt{CPACR_EL1}, \texttt{CPTR_EL2}, and \texttt{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{Scalar} and \texttt{Vector}

\textbf{Scalar}

\begin{verbatim}
SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L);   Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
\end{verbatim}

\textbf{Vector}

\begin{verbatim}
SQDMULL, SQDMULL2 (by element)
\end{verbatim}
Vector

SQDMULL[2] <Vd>,<Ta>,<Vn>,<Tb>,<Vm>[<index>]

integer idxdsz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Va> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:
Restricted to V0-V15 when element size $<Ts>$ is H.

$<Ts>$ is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>00</th>
<th>RESERVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
<td></td>
</tr>
</tbody>
</table>

$<$index$>$ is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>00</th>
<th>RESERVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>H:L:M</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
<td></td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(idxdsizw) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMULL, SQDMULL2 (vector)**

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit `FPSCR.QC` is set.

The `SQDMULL` instruction extracts each source vector from the lower half of each source register, while the `SQDMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Scalar**

```
SQDMULL <Va><d>, <Vb><n>, <Vb><m>
```

```plaintext
d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;

esize = 8 << UInt(size);

datasize = esize;
elements = 1;
part = 0;
```

**Vector**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Vector**

```
SQDMULL(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```plaintext
d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;

esize = 8 << UInt(size);

datasize = 64;

part = UInt(Q);
elements = datasize DIV esize;
```

**Assembler Symbols**

2  
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

---

`<Vd>`  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>`  
Is an arrangement specifier, encoded in “size”:
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is an arrangement specifier, encoded in “size-Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
</tr>
</tbody>
</table>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
if sat then FPSR.QC = '1';
V[d] = result;
```
Signed saturating Negate. This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | size | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | Rn | Rd |
| U  |

**Scalar**

\[
\text{SQNEG} <V><d>, <V><n> \]

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');

**Vector**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | size | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | Rn | Rd |
| U  |

**Vector**

\[
\text{SQNEG} <Vd>.<T>, <Vn>.<T> \]

integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

**Assembler Symbols**

\[<V>\] Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\[<d>\] Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\[<n>\] Is the number of the SIMD&FP source register, encoded in the "Rn" field.
\[<Vd>\] Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\[<T>\] Is an arrangement specifier, encoded in “size:Q”:
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```
SQRDMLAH (by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPRS.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(Armv8.1)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 size | L | M | Rm | 1 | 1 | 0 | 1 | H | 0 | Rn | Rd

Scalar

SQRDMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

if !HasQRDMLAHExt() then UNDEFINED;

integer idxdsz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector

(Armv8.1)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 size | L | M | Rm | 1 | 1 | 0 | 1 | H | 0 | Rn | Rd

SQRDMLAH (by element)
Vector

SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:M:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in "size:L:H:M":

SQRDMLAH (by element)
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;
element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  element3 = SInt(Elem[operand3, e, esize]);
  if sub_op then
    accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
  else
    accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
  (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
  if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
Scalar
(ARMv8.1)
```

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
s 0 1 1 1 1 1 0 |   Rm | 1 0 0 0 0 0 1 | Rn | Rd
```

Scalar

```
SQRDMLAH <V><d>, <V><n>, <V><m>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' | size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

```
Vector
(ARMv8.1)
```

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Q 0 1 0 1 1 1 0 |   Rm | 1 0 0 0 0 0 1 | Rn | Rd
```

Vector

```
SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' | size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

```
<V> Is a width specifier, encoded in “size”:
```

SQRDMLAH (vector)
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLSH (by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(ARMv8.1)

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Scalar

```
SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

if !HaveQRDMLAHExt() then UNDEFINED;

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
```

```
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
```

```
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

(ARMv8.1)

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Vector

\[
\text{SQRDMLSH} \quad <\text{Vd}, <\text{T}, <\text{Vn}>. <\text{T}, <\text{Vm}>. <\text{T}>[<\text{index}>] \]

if \( \text{HaveQRDMLAHExt}() \) then UNDEFINED;

integer idxdsize = if \( H == '1' \) then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if \( Q == '1' \) then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = \( \text{S} == '1' \);

Assembler Symbols

\( <\text{V}> \quad \text{Is a width specifier, encoded in \"size\":} \)

<table>
<thead>
<tr>
<th>size</th>
<th>( &lt;\text{V}&gt; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>( H )</td>
</tr>
<tr>
<td>10</td>
<td>( S )</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\( <\text{d}> \quad \text{Is the number of the SIMD&FP destination register, encoded in the \"Rd\" field.} \)

\( <\text{n}> \quad \text{Is the number of the first SIMD&FP source register, encoded in the \"Rn\" field.} \)

\( <\text{Vd}> \quad \text{Is the name of the SIMD&FP destination register, encoded in the \"Rd\" field.} \)

\( <\text{T}> \quad \text{Is an arrangement specifier, encoded in \"size:Q\":} \)

<table>
<thead>
<tr>
<th>size</th>
<th>( &lt;\text{T}&gt; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>( \times )</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1 4H</td>
</tr>
<tr>
<td>10</td>
<td>0 2S</td>
</tr>
<tr>
<td>10</td>
<td>1 4S</td>
</tr>
<tr>
<td>11</td>
<td>( \times )</td>
</tr>
</tbody>
</table>

\( <\text{Vn}> \quad \text{Is the name of the first SIMD&FP source register, encoded in the \"Rn\" field.} \)

\( <\text{Vm}> \quad \text{Is the name of the second SIMD&FP source register, encoded in \"size:M:Rm\":} \)

<table>
<thead>
<tr>
<th>size</th>
<th>( &lt;\text{Vm}&gt; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size \( <\text{Ts}> \) is H.

\( <\text{Ts}> \quad \text{Is an element size specifier, encoded in \"size\":} \)

<table>
<thead>
<tr>
<th>size</th>
<th>( &lt;\text{Ts}&gt; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>( H )</td>
</tr>
<tr>
<td>10</td>
<td>( S )</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\( <\text{index}> \quad \text{Is the element index, encoded in \"size:L:H:M\":} \)
### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  element3 = SInt(Elem[operand3, e, esize]);
  if sub_op then
    accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
  else
    accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
  (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
  if sat then FPSR.QC = '1';

V[d] = result;
```

---

**Operation**

CheckFPAdvSIMDEnabled64();

```c
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  element3 = SInt(Elem[operand3, e, esize]);
  if sub_op then
    accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
  else
    accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
  (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
  if sat then FPSR.QC = '1';

V[d] = result;
```
**SQRDMLSH (vector)**

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, \( FSR.QC \), is set if saturation occurs.

Depending on the settings in the \( CPACR_EL1, CPTR_EL2, \) and \( CPTR_EL3 \) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

**Scalar**

**Scalar**

**Scalar**

**Scalar**

**Scalar**

**Scalar**

**Scalar**

**Scalar**

**Scalar**

### Vector

**Vector**

**Vector**

**Vector**

**Vector**

### Assembler Symbols

\(<V>\) Is a width specifier, encoded in “size”: 
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m>
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 \ll (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 \ll esize) - 2 \times (element1 \times element2) + rounding_const);
    else
        accum = ((element3 \ll esize) + 2 \times (element1 \times element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00ber8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQRDMULH (by element)**

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see $\text{SQDMULH}$.

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit $\text{FPSR.QC}$ is set.

Depending on the settings in the $\text{CPACR_EL1}$, $\text{CPTR_EL2}$, and $\text{CPTR_EL3}$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | size | L  | M  | Rm | 1   | 1   | 0   | 1   | H  | 0  | Rn | Rd |

**op**

### Vector

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**op**

---

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L);   Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
Vector

SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean round = (op == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<n> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in “size:L:H:M”:
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SQUAD.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | size | 1  | Rm | 1  | 0  | 1  | 1  | 0  | 1  | Rn | 1  | Rd |
| U  |

Scalar

SQRDMULH <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 0  | size | 1  | Rm | 1  | 0  | 1  | 1  | 0  | 1  | Rn | 1  | Rd |
| U  |

Vector

SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPA dvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes to the vector the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see \textit{SQRSHL}.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \textit{FPSR}.QC is set.

Depending on the settings in the \textit{CPACR_EL1}, \textit{CPTR_EL2}, and \textit{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \textit{Scalar} and \textit{Vector}.

**Scalar**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  |
| U  | R  | S  |
```

**Vector**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  |
| U  | R  | S  |
```

**Assembler Symbols**

\(<V>\) Is a width specifier, encoded in “size”: 

\[
\text{integer } d = \text{UInt}(Rd);
\text{integer } n = \text{UInt}(Rn);
\text{integer } m = \text{UInt}(Rm);
\text{integer esize} = 8 \ll \text{UInt}(\text{size});
\text{integer dataszize} = \text{esize};
\text{integer elements} = 1;
\text{boolean unsigned} = (U == '1');
\text{boolean rounding} = (R == '1');
\text{boolean saturating} = (S == '1');
\] 

\text{if } S == '0' \&\& \text{size} != '11' \text{ then UNDEFINED;
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];

bits(datasize) operand2 = V[m];

bits(datasize) result;

integer round_const = 0;

integer shift;

integer element;

boolean sat;

for e = 0 to elements-1

    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
        else
            Elem[result, e, esize] = element<esize-1:0>;
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SQRSHRN.

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 1  | 0  | 0  | 1  | 1  | Rn  | Rd  |

U: immh: op

**Scalar**

SQRSHRN <Vb><d>, <Va><n>, #<shift>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

to do:
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

**Vector**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 1  | 0  | 0  | 1  | 1  | Rn  | Rd  |

U: immh: op

**Vector**

SQRSHRN[2] <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```
2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb>

Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>

Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va>

Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift>

For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
  element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
  (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
  if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRSHRUN, SQRSHRUN2

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see SQRSHRUN.

The SQRSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | ! = 0000 | immh | 1  | 0  | 0  | 0  | 1  | Rn | Rd |

Scalar

SQRSHRUN <Vb><d>, <Va><n>, #<shift>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | ! = 0000 | immh | 1  | 0  | 0  | 0  | 1  | 1  | Rn | Rd |

Vector

SQRSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE{asimdimm};
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```
Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td><strong>SEE Advanced SIMD modified immediate</strong></td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td><strong>SEE Advanced SIMD modified immediate</strong></td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td><strong>SEE Advanced SIMD modified immediate</strong></td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = n;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
SQSHL (immediate)

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 0  | 1  | 1  | 0  | 1  | Rn  | Rd  |
| U  | immh | op |

Scalar

SQSHL <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 0  | 1  | 1  | 0  | 1  | Rn  | Rd  |
| U  | immh | op |
Vector

SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = Uint(Rd);
integer n = Uint(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = Uint(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE;  dst_unsigned = TRUE;
```

Assembler Symbols

<\V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>\V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(Uint(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(Uint(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(Uint(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(Uint(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(Uint(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(Uint(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(Uint(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(Uint(immh:immb)-64)</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQSHL (register)

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see \textit{SQRSHL}.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \textit{FPSR}.QC is set.

Depending on the settings in the \textit{CPACR_EL1}, \textit{CPTR_EL2}, and \textit{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{Scalar} and \texttt{Vector}

### Scalar

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>size</td>
<td>1</td>
<td></td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\texttt{SQSHL <V<d>, <V<n>, <V<m>}

integer d = \texttt{UInt}(Rd);
n = \texttt{UInt}(Rn);
m = \texttt{UInt}(Rm);

integer esize = 8 << \texttt{UInt}(size);
integer datasize = esize;
integer elements = 1;

if \texttt{S == '0'} && size != '11' then UNDEFINED;

### Vector

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>size</td>
<td>1</td>
<td></td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\texttt{SQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

integer d = \texttt{UInt}(Rd);
n = \texttt{UInt}(Rn);
m = \texttt{UInt}(Rm);

if size\texttt{Q == '110'} then UNDEFINED;

integer esize = 8 << \texttt{UInt}(size);
integer datasize = if \texttt{Q == '1'} then 128 else 64;
integer elements = datasize \texttt{DIV esize};

\texttt{Assembler Symbols}

\texttt{<V>} Is a width specifier, encoded in “size”:
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m>
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00be9f_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see $UQRSHLU$.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>!= 0000</th>
<th>immh</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

U | op | immh |

Scalar

$SQSHLU \langle V \rangle <d>, \langle V \rangle <n>, \#<shift>$

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
if immh == '0000' then UNDEFINED;
integer esize = 8 << $\text{HighestSetBit}(\text{immh})$;
integer datasize = esize;
integer elements = 1;
integer shift = $\text{UInt}(\text{immh}:\text{immb}) - \text{esize}$;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>!= 0000</th>
<th>immh</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

U | op | immh |
Vector

\[ \text{SQSHLU} \ <Vd>., <T>, \ <Vn>, .<T>, \ #<\text{shift}> \]

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;

\text{case op:U of}
  \text{when '00' UNDEFINED;}
  \text{when '01' src_unsigned = FALSE; dst_unsigned = TRUE;}
  \text{when '10' src_unsigned = FALSE; dst_unsigned = FALSE;}
  \text{when '11' src_unsigned = TRUE; dst_unsigned = TRUE;}
\end{verbatim}

Assembler Symbols

\(<V>\) Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0 \ SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>1 \ 8B</td>
</tr>
<tr>
<td>001x</td>
<td>0 \ 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 \ 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 \ 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 \ 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 \ RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 \ 2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<\text{shift}>\) For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;\text{shift}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;\text{shift}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSHRN, SQSHRN2**

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see [SQSXRN](#).

The SQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 1   | 1   | 1   | 1   | 1   | 0   | !=0000 | immh | 1 | 0 | 0 | 1 | 0 | 1 | Rd | Rn | op | immh |
| U   |

**Scalar**

\[
\text{SQSHRN} \langle V_b > \langle d >, \langle V_a > \langle n >, \# < \text{shift} >
\]

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | Q  | 0   | 1   | 1   | 1   | 1   | 0   | !=0000 | immh | 1 | 0 | 0 | 1 | 0 | 1 | Rd | Rn | op | immh |
| U   |

**Vector**

\[
\text{SQSHRN}\{2\} \langle V_d > \langle T_b >, \langle V_n > \langle T_a >, \# < \text{shift} >
\]

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```
Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb>

Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>

Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va>

Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift>

For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
CheckFPAdvSIMDEnabled64();
bcds(\text{datasize} \times 2) \text{ operand } = V[n];
bcds(\text{datasize}) \text{ result;}
\text{integer round\_const } = \text{ if round then } (1 \ll (\text{shift} - 1)) \text{ else } 0;
\text{integer element;}
\text{boolean sat;}

\text{for } e = 0 \text{ to elements\textendash}1
\begin{align*}
\text{element } &= \left(\text{Int}(\text{Elem}[\text{operand}, e, 2 \times \text{esize}], \text{unsigned}) + \text{round\_const}\right) \gg \text{shift};
\text{SatQ}(\text{Elem}[\text{result}, e, \text{esize}], \text{sat}) &= \text{SatQ}(\text{element}, \text{esize}, \text{unsigned});
\text{if sat then FPSR.QC } = '1';
\end{align*}
\text{Vpart}[d, \text{part}] = \text{result;}

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see SQSHRUN.

The SQSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{Scalar} and \texttt{Vector}.

**Scalar**

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & Rn & Rd \\
\end{array}
\]

**Scalar**

\[
\text{SQSHRUN} \langle Vb \rangle \langle d \rangle, \langle Va \rangle \langle n \rangle, \# \langle \text{shift} \rangle
\]

\[
\begin{array}{l}
\text{integer } d = \text{\texttt{UInt}}(Rd);
\text{integer } n = \text{\texttt{UInt}}(Rn);
\end{array}
\]

\[
\begin{array}{l}
\text{if immh == '0000' then UNDEFINED; }
\text{if immh<3> == '1' then UNDEFINED; }
\text{integer esize = 8 \times \texttt{HighestSetBit}(immh); }
\text{integer datatize = esize; }
\text{integer elements = 1; }
\text{integer part = 0; }
\end{array}
\]

\[
\begin{array}{l}
\text{integer shift = (2 \times \text{esize}) - \text{\texttt{UInt}}(immh:immb); }
\text{boolean round = (op == '1'); }
\end{array}
\]

**Vector**

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & Q & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & Rn & Rd \\
\end{array}
\]

**Vector**

\[
\begin{array}{l}
\text{SQSHRUN(2) \langle Vd \rangle \langle Tb \rangle, \langle Vn \rangle \langle Ta \rangle, \# \langle \text{shift} \rangle}
\end{array}
\]

\[
\begin{array}{l}
\text{integer } d = \text{\texttt{UInt}}(Rd);
\text{integer } n = \text{\texttt{UInt}}(Rn);
\end{array}
\]

\[
\begin{array}{l}
\text{if immh == '0000' then } \texttt{SEE(asimdimm);} \\
\text{if immh<3> == '1' then UNDEFINED; } \\
\text{integer esize = 8 \times \texttt{HighestSetBit}(immh); } \\
\text{integer datatize = 64; } \\
\text{integer elements = datatize DIV esize; } \\
\end{array}
\]

\[
\begin{array}{l}
\text{integer shift = (2 \times \text{esize}) - \text{\texttt{UInt}}(immh:immb); } \\
\text{boolean round = (op == '1');}
\end{array}
\]
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<\textit{Vd}> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\textit{Tb}> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>\textit{&lt;Tb&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<\textit{Vn}> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<\textit{Ta}> Is an arrangement specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>\textit{&lt;Ta&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<\textit{Vb}> Is the destination width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>\textit{&lt;Vb&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<\textit{d}> Is the number of the SIMD&FP destination register, in the "Rd" field.

<\textit{Va}> Is the source width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>\textit{&lt;Va&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<\textit{n}> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\textit{shift}> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>\textit{&lt;shift&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>\textit{&lt;shift&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```
SIGNED SATURATING SUBTRACT

This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \( \text{FPSR}.QC \) is set. Depending on the settings in the \( \text{CPACR_EL1}, \text{CPTR_EL2}, \) and \( \text{CPTR_EL3} \) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

\[
\begin{array}{cccccccccccccc}
\hline
0 & 1 & 0 & 1 & 1 & 1 & 0 & \text{size} & 1 & \text{Rm} & 0 & 0 & 1 & 0 & 1 & 1 & \text{Rn} & \text{Rd} \\
\end{array}
\]

U

**Scalar**

\[
\text{SQSUB} <V><d>, <V><n>, <V><m>
\]

```python
integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rm);
integer esize = 8 << \text{UInt}(\text{size});
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

**Vector**

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & Q & 0 & 1 & 1 & 1 & 0 & \text{size} & 1 & \text{Rm} & 0 & 0 & 1 & 0 & 1 & 1 & \text{Rn} & \text{Rd} \\
\end{array}
\]

U

**Vector**

\[
\text{SQSUB} <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
\]

```python
integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

**Assembler Symbols**

- **<V>** Is a width specifier, encoded in "\text{size}"
  
  \[
  \begin{array}{ccc}
  \text{size} & <V> \\
  00 & B \\
  01 & H \\
  10 & S \\
  11 & D \\
  \end{array}
  \]

- **<d>** Is the number of the SIMD&FP destination register, in the "\text{Rd}" field.
- **<n>** Is the number of the first SIMD&FP source register, encoded in the "\text{Rn}" field.
- **<m>** Is the number of the second SIMD&FP source register, encoded in the "\text{Rm}" field.
- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "\text{Rd}" field.
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
if sat then FPSR.QC = '1';
V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQXTN, SQXTN2**

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \( \text{FPSR.QC} \) is set. The \text{SQXTN} instruction writes the vector to the lower half of the destination register and clears the upper half; while the \text{SQXTN2} instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the \text{CPACR_EL1}, \text{CPTR_EL2}, and \text{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>size</th>
<th>Rd</th>
<th>Rn</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 1 0 0 1 0</td>
<td>1 1 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Scalar

\[
\text{SQXTN} \ <Vb><d>, \ <Va><n>
\]

- integer \( d = \text{UInt}(Rd) \);
- integer \( n = \text{UInt}(Rn) \);
- if size == '11' then UNDEFINED;
- integer \( \text{esize} = 8 \times \text{UInt}(\text{size}) \);
- integer \( \text{datasize} = \text{esize} \);
- integer \( \text{part} = 0 \);
- integer \( \text{elements} = 1 \);
- boolean \( \text{unsigned} = (U == '1') \);

### Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>size</th>
<th>Rd</th>
<th>Rn</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 1 0 0 1 0</td>
<td>1 1 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Vector

\[
\text{SQXTN(2)} \ <Vd>.<Tb>, \ <Vn>.<Ta>
\]

- integer \( d = \text{UInt}(Rd) \);
- integer \( n = \text{UInt}(Rn) \);
- if size == '11' then UNDEFINED;
- integer \( \text{esize} = 8 \times \text{UInt}(\text{size}) \);
- integer \( \text{datasize} = 64 \);
- integer \( \text{part} = \text{UInt}(Q) \);
- integer \( \text{elements} = \text{datasize} \times \text{DIV esize} \);
- boolean \( \text{unsigned} = (U == '1') \);

### Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”: 

SQXTN, SQXTN2
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQXTUN, SQXTUN2

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit $FP SR.QC$ is set.

The SQXTUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |

Scalar

```plaintext
SQXTUN <Vb><d>, <Va><n>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasource = esize;
integer part = 0;
integer elements = 1;
```

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |

Vector

```plaintext
SQXTUN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasource = 64;
integer part =(UInt(Q));
integer elements = datasource DIV esize;
```

Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;Vd&gt;</th>
<th>Is the name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Tb&gt;</td>
<td>Is an arrangement specifier, encoded in “size:Q”:</td>
</tr>
<tr>
<td>size</td>
<td>Q</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Element[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v3.0.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Integer d = UInt(Rd);
Integer n = UInt(Rn);
Integer m = UInt(Rm);
If size == '11' then UNDEFINED;
Integer esize = 8 << UInt(size);
Integer datasize = if Q == '1' then 128 else 64;
Integer elements = datasize DIV esize;
Boolean unsigned = (U == '1');

Assembler Symbols

<Vd> is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> is an arrangement specifier, encoded in “size-Q”;

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
Bits(datasize) operand1 = V[n];
Bits(datasize) operand2 = V[m];
Bits(datasize) result;
Integer element1;
Integer element2;
For e = 0 to elements-1
   element1 = Int(Elem[operand1, e, esize], unsigned);
   element2 = Int(Elem[operand2, e, esize], unsigned);
   Elem[result, e, esize] = (element1 + element2 + 1)<<esize:1>;
V[d] = result;
SRI

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

The following figure shows an example of the operation of shift right by 3 for an 8-bit vector element.

![Vn.B[7] before operation](image)

![Vn.B[7] after operation](image)

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

![Scalar encoding](image)

SRI `<V><d>, <V><n>, #<shift>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
```

### Vector

![Vector encoding](image)

SRI `<V>`
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

if immh == '0000' then \texttt{SEE(asimdimm)};
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 \ll \texttt{HighestSetBit}(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize \div esize;
integer shift = (esize * 2) - \texttt{UInt}(immh:immb);

Assembler Symbols

\texttt{<V>} Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

\texttt{<d>} Is the number of the SIMD&FP destination register, in the "Rd" field.

\texttt{<n>} Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<T>} Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
</tbody>
</table>

\texttt{<Vn>} Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\texttt{<shift>} For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-\texttt{UInt}(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>\texttt{SEE Advanced SIMD modified immediate}</td>
</tr>
<tr>
<td>0001</td>
<td>(16-\texttt{UInt}(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-\texttt{UInt}(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-\texttt{UInt}(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-\texttt{UInt}(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

\texttt{CheckFPAvSIMDEnabled64}();
\texttt{bits(datasize) operand = V[n];}
\texttt{bits(datasize) operand2 = V[d];}
\texttt{bits(datasize) result;}
\texttt{bits(esize) mask = \texttt{LSR} \texttt{(Ones}(esize), shift);}
\texttt{bits(esize) shifted;}

for e = 0 to elements-1
  \texttt{shifted = \texttt{LSR}(Elem[operand, e, esize], shift);}
  \texttt{Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;}
\texttt{V[d] = result;}
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00beta8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Rounding Shift Left (Register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see SSHL. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>R</td>
<td>S</td>
<td>size</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Vector

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>R</td>
<td>S</td>
<td>size</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>Rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Assembler Symbols

- `<V>` Is a width specifier, encoded in "size";

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
        if saturating then
            (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
            if sat then FPSR.QC = '1';
            else
                Elem[result, e, esize] = element<esize-1:0>;
        else
            Elem[result, e, esize] = element<esize-1:0>;
    else
        V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see S SHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>! = 0000</td>
<td>immb</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>immh</td>
<td></td>
<td>o1</td>
<td>o0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Scalar

```
SRSHR <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>! = 0000</td>
<td>immb</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>immh</td>
<td></td>
<td>o1</td>
<td>o0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Vector

```
SRSHR <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

< V > Is a width specifier, encoded in “immh”:
### <immh> <Vn> <T> <shift>

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td><strong>SEE Advanced SIMD modified immediate</strong></td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>0xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

#### <<Vn>><T>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

#### <T>

Is an arrangement specifier, encoded in “immh:Q”:

#### <shift>

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td><strong>SEE Advanced SIMD modified immediate</strong></td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

#### Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

---

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right
shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the
values in this instruction are signed integer values. The results are rounded. For truncated results, see SSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an
attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

Scalar $SRSRA <V><d>$, $<V><n>$, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if imm<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

Vector $SRSRA <Vd>.<T>$, $<Vn>.<T>$, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

$<V>$ Is a width specifier, encoded in “immh”:
**immh** <V>

| 0xxx | RESERVED       |
| 1xxx | D              |

**<d>**

Is the number of the SIMD&FP destination register, in the "Rd" field.

**<n>**

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vd>**

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>**

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<Vn>**

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<shift>**

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
exticient element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB.

- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 1 1 1 1
CRm   opc
```

System

```
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if HaveSBExt() && CRm<3:0> == '0000' then
      op = MemBarrierOp_SB;
    else
      UNDEFINED;
  end;

case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
  otherwise
    if CRm<3:2> == '01' then
      op = MemBarrierOp_PSSBB;
    elsif CRm<3:2> == '00' && opc == '00' then
      op = MemBarrierOp_SSBB;
    elsif HaveSBExt() && CRm<3:2> == '00' && opc == '11' then
      op = MemBarrierOp_SB;
    else
      types = MBReqTypes_All;
      domain = MBReqDomain_FullSystem;
end;
```
case op of
  when MemBarrierOp_DSB
      DataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMB
      DataMemoryBarrier(domain, types);
  when MemBarrierOp_ISB
      InstructionSynchronizationBarrier();
  when MemBarrierOp_SSBB
      SpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBB
      SpeculativeSynchronizationBarrierToPA();
  when MemBarrierOp_SB
      SpeculationBarrier();
SSHL

Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value
by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector,
and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see
SSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an
attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Scalar

SSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Vector

SSHL <Vd><T>, <Vn><T>, <Vm><T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

SSHL
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m>
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
   shift = SInt(Elem[operand2, e, esize]<7:0>);
   if rounding then
      round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
   element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
   if saturating then
      (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
      if sat then FPSR.QC = '1';
   else
      Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSHLL, SSHLL2

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register, while the SSHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias SXTL, SXTL2.

Vector

\[
\text{SSHLL}[^2] <Vd>, <Ta>, <Vn>, <Tb> \text{, } #<\text{shift}>
\]

integer \( d = \text{UInt}(Rd); \)
integer \( n = \text{UInt}(Rn); \)

if \( \text{immh} \) == '0000' then \( \text{SEE(asimdimm)}; \)
if \( \text{immh}<3> \) == '1' then \( \text{UNDEFINED}; \)
integer esize = 8 \( \ll \) HighestSetBit(immh);
integer datasize = 64;
integer part = \( \text{UInt}(Q); \)
integer elements = datasize \( \div \) esize;

integer shift = \( \text{UInt}(\text{immh:immb}) \) - esize;
boolean unsigned = (U == '1');

Assembler Symbols

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

\(<Vd>\)
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Ta>\)
Is an arrangement specifier, encoded in “immh”:

\[
\begin{array}{c|c}
\text{immh} & \text{<Ta>} \\
0000 & \text{SEE Advanced SIMD modified immediate} \\
0001 & 8H \\
001x & 4S \\
01xx & 2D \\
1xxx & \text{RESERVED}
\end{array}
\]

\(<Vn>\)
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\)
Is an arrangement specifier, encoded in “immh:Q”:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
</tr>
</tbody>
</table>

\(<\text{shift}>\)
Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:
### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTL, SXTL2</td>
<td>immh == '000' &amp;&amp; BitCount(immh) == 1</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSHR

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

SSHR <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

SSHR <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdim);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “immh”: 
**immh**  &lt;V&gt;  Table

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

&lt;d&gt;  Is the number of the SIMD&FP destination register, in the "Rd" field.

&lt;n&gt;  Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

&lt;Vd&gt;  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

&lt;T&gt;  Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

&lt;Vn&gt;  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

&lt;shift&gt;  For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 &lt;&lt; (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) &gt;&gt; shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element&lt;esize-1:0&gt;;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SSRA

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 1 8
Rn Rd
U immh o1 o0
```

Scalar

\[
\text{SSRA } <V><d>, <V><n>, #<shift>}
\]

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

```
0 1 1 1 1 1 0 0 0 0 1 0 1 0 1 8
Rn Rd
U immh o1 o0
```

Vector

\[
\text{SSRA } <Vd><T>, <Vn><T>, #<shift>}
\]

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

\[
<V>
\]

Is a width specifier, encoded in “immh”:
**<D>**
Is the number of the SIMD&FP destination register, in the "Rd" field.

**<n>**
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vd>**
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>**
Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

**<Vn>**
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<shift>**
For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>4H</td>
</tr>
<tr>
<td>01xx</td>
<td>2S</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

---

**Operation**

```cpp
CheckFPAdvSIMDEnabled64();
bots(datasize) operand = V[n];
bots(datasize) operand2;
bots(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSUBL, SSUBL2

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The SSUBL instruction extracts each source vector from the lower half of each source register, while the SSUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Assembler Symbols**

2  
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

$<Vd>$  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Ta>$  
Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;Ta&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<Vn>$  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Tb>$  
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>$&lt;Tb&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<Vm>$  
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register, while the SSUBW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>1</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

Three registers, not all the same type

SSUBW{2} <Vd>,<Ta>, <Vn>,<Ta>, <Vm>.<Tb>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th></th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>

Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
   element1 = Int(IntElem[operand1, e, 2*esize], unsigned);
   element2 = Int(IntElem[operand2, e, esize], unsigned);
   if sub_op then
      sum = element1 - element2;
   else
      sum = element1 + element2;
   Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One register (opcode == 0111)

ST1 { <Vt>.<T> }, [<Xn|SP>]

Two registers (opcode == 1010)

ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

Three registers (opcode == 0110)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

Four registers (opcode == 0010)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td>Rm</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
One register, immediate offset (Rm == 11111 && opcode == 0111)

\[
\text{ST1 } \{ <Vt>.<T> \}, [<Xn|SP>], \text{ imm}
\]

One register, register offset (Rm != 11111 && opcode == 0111)

\[
\text{ST1 } \{ <Vt>.<T> \}, [<Xn|SP>], <Xm>
\]

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

\[
\text{ST1 } \{ <Vt>.<T>, <Vt2>.<T> \}, [<Xn|SP>], \text{ imm}
\]

Two registers, register offset (Rm != 11111 && opcode == 1010)

\[
\text{ST1 } \{ <Vt>.<T>, <Vt2>.<T> \}, [<Xn|SP>], <Xm>
\]

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

\[
\text{ST1 } \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>], \text{ imm}
\]

Three registers, register offset (Rm != 11111 && opcode == 0110)

\[
\text{ST1 } \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>], <Xm>
\]

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

\[
\text{ST1 } \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> \}, [<Xn|SP>], \text{ imm}
\]

Four registers, register offset (Rm != 11111 && opcode == 0010)

\[
\text{ST1 } \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> \}, [<Xn|SP>], <Xm>
\]

integer \( t = \text{ UInt}(Rt); \)
integer \( n = \text{ UInt}(Rn); \)
integer \( m = \text{ UInt}(Rm); \)
boolean \( wback = \text{ TRUE}; \)

Assembler Symbols

\(<Vt>\) Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

\(<T>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vt2>\) Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

\(<Vt3>\) Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

\(<Vt4>\) Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{imm}>\) For the one register, immediate offset variant: is the post-index immediate offset, encoded in "Q":
For the two registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#8</td>
</tr>
<tr>
<td>1</td>
<td>#16</td>
</tr>
</tbody>
</table>

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:  

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

<Xm>  
Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            endif
    endif
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
endif
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

**No offset**

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | x   | x   | 0   | S   | size | Rn  | Rt  |
| L   | R   | opcode |

**8-bit (opcode == 000)**

\[
\text{ST1} \{<\text{Vt}.B>[<\text{index}>], [<\text{Xn}|\text{SP}>]}
\]

**16-bit (opcode == 010 && size == x0)**

\[
\text{ST1} \{<\text{Vt}.H>[<\text{index}>], [<\text{Xn}|\text{SP}>]}
\]

**32-bit (opcode == 100 && size == 00)**

\[
\text{ST1} \{<\text{Vt}.S>[<\text{index}>], [<\text{Xn}|\text{SP}>]}
\]

**64-bit (opcode == 100 && S == 0 && size == 01)**

\[
\text{ST1} \{<\text{Vt}.D>[<\text{index}>], [<\text{Xn}|\text{SP}>]}
\]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

**Post-index**

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 0   | 1   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | x   | x   | 0   | S   | size | Rn  | Rt  |
| L   | R   | opcode |

8-bit, immediate offset (Rm == 11111 && opcode == 000)

\[
\text{ST1 \{ \langle Vt \rangle.B \}[\langle index \rangle], \langle Xn|SP \rangle, \#1}
\]

8-bit, register offset (Rm != 11111 && opcode == 000)

\[
\text{ST1 \{ \langle Vt \rangle.B \}[\langle index \rangle], \langle Xn|SP \rangle, \langle Xm \rangle}
\]

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

\[
\text{ST1 \{ \langle Vt \rangle.H \}[\langle index \rangle], \langle Xn|SP \rangle, \#2}
\]

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

\[
\text{ST1 \{ \langle Vt \rangle.H \}[\langle index \rangle], \langle Xn|SP \rangle, \langle Xm \rangle}
\]

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

\[
\text{ST1 \{ \langle Vt \rangle.S \}[\langle index \rangle], \langle Xn|SP \rangle, \#4}
\]

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

\[
\text{ST1 \{ \langle Vt \rangle.S \}[\langle index \rangle], \langle Xn|SP \rangle, \langle Xm \rangle}
\]

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

\[
\text{ST1 \{ \langle Vt \rangle.D \}[\langle index \rangle], \langle Xn|SP \rangle, \#8}
\]

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

\[
\text{ST1 \{ \langle Vt \rangle.D \}[\langle index \rangle], \langle Xn|SP \rangle, \langle Xm \rangle}
\]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

**Assembler Symbols**

\(<Vt>\) Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

\(<index>\) For the 8-bit variant: is the element index, encoded in "Q:S:size".

\(<index>\) For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

\(<index>\) For the 32-bit variant: is the element index, encoded in "Q:S".

\(<index>\) For the 64-bit variant: is the element index, encoded in "Q".

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<Xm>\) Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);       // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);   // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);        // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);          // D[0-1]
      scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvS1MEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

**No offset**

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | size |     |     |     |     |     |     |     |     |     |     |

L

opcode

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

**Post-index**

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | size |     |     |     |     |     |     |     |     |     |     |

L

opcode

Immediate offset (Rm == 1111)

ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 1111)

ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

**Assembler Symbols**

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

```c
Operation CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!wback && n == 31);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

offs = Zeros();
for r = 0 to rpt-1
  for e = 0 to elements-1
    tt = (t + r) MOD 32;
    for s = 0 to selem-1
      rval = V[tt];
      if memop == MemOp_LOAD then
        Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
        V[tt] = rval;
      else // memop == MemOp_STORE
        Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
      offs = offs + ebytes;
      tt = (tt + 1) MOD 32;
  if wback then
    if m != 31 then
      offs = X[m];
    if n == 31 then
      SP[] = address + offs;
    else
      X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

8-bit (opcode == 000)

ST2 \{ <Vt>.B, <Vt2>.B \}[<index>], [<Xn|SP>]

16-bit (opcode == 010 && size == x0)

ST2 \{ <Vt>.H, <Vt2>.H \}[<index>], [<Xn|SP>]

32-bit (opcode == 100 && size == 00)

ST2 \{ <Vt>.S, <Vt2>.S \}[<index>], [<Xn|SP>]

64-bit (opcode == 100 && S == 0 && size == 01)

ST2 \{ <Vt>.D, <Vt2>.D \}[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
8-bit, immediate offset (Rm == 11111 && opcode == 000)
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2

8-bit, register offset (Rm != 11111 && opcode == 000)
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
      scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index

(ARMv8.5)

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & \text{imm9} & 0 & 1 & \text{Xn} & \text{(1)(1)(1)(1)(1)}
\end{array}
\]

Post-index

ST2G [<Xn|SP>], #<simm>

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
boolean zero_data = FALSE;

Pre-index

(ARMv8.5)

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & \text{imm9} & 1 & 1 & \text{Xn} & \text{(1)(1)(1)(1)(1)}
\end{array}
\]

Pre-index

ST2G [<Xn|SP>, #<simm>]

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
boolean zero_data = FALSE;

Signed offset

(ARMv8.5)

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & \text{imm9} & 1 & 0 & \text{Xn} & \text{(1)(1)(1)(1)(1)}
\end{array}
\]

Signed offset

ST2G [<Xn|SP>{, #<simm>})

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
boolean zero_data = FALSE;
Assembler Symbols

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;
bits(4) tag;
SetNotTagCheckedInstruction(TRUE);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
if zero_data then
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8*TAG_GRANULE);
    Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(8*TAG_GRANULE);
tag = AllocationTagFromAddress(address);
MemTag[address] = tag;
MemTag[address+TAG_GRANULE] = tag;
if writeback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```
ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 0   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | size | Rn  | Rt  |

L     opcode

No offset

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | 0   | Rm  | 0   | 1   | 0   | 0   | size | Rn  | Rt  |

L     opcode

Immediate offset (Rm == 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

Assembler Symbols

-Vt-
Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

-T-
Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

-Vt2-
Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

-Vt3-
Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<table>
<thead>
<tr>
<th>0</th>
<th>#24</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

Is the post-index immediate offset, encoded in "Q":

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;
integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

if size:Q == '110' && selem != 1 then UNDEFINED;
```

ST3 (multiple structures)
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!wback && n == 31);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

offs = Zeros();
for r = 0 to rpt-1
  for e = 0 to elements-1
    tt = (t + r) MOD 32;
    for s = 0 to selem-1
      rval = V[tt];
      if memop == MemOp_LOAD then
        Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
        V[tt] = rval;
      else // memop == MemOp_STORE
        Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
      offs = offs + ebytes;
    tt = (tt + 1) MOD 32;

if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | S | size | Rn | Rt |
|-----------------------------------------------|-----------------|
| L | R | opcode |

8-bit (opcode == 001)


16-bit (opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]

32-bit (opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]

64-bit (opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | Rm | x | x | 1 | S | size | Rn | Rt |
|-----------------------------------------------|-----------------|
| L | R | opcode |
8-bit, immediate offset (Rm == 11111 && opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3

8-bit, register offset (Rm != 11111 && opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
Shared Decode

```plaintext
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);       // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);          // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);            // D[0-1]
      scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
  integer datasize = if Q == '1' then 128 else 64;
  integer esize = 8 << scale;
```

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index.

No offset

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 Q | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

L

opcode

Rn

Rt

No offset

\[
\text{ST4} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> \}, [<Xn|SP>]
\]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 Q | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

L

opcode

Rm

Rn

Rt

Immediate offset (Rm == 11111)

\[
\text{ST4} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> \}, [<Xn|SP>], <imm>
\]

Register offset (Rm != 11111)

\[
\text{ST4} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> \}, [<Xn|SP>], <Xm>
\]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

Assembler Symbols

\(<Vt>\) Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

\(<T>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vt2>\) Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

\(<Vt3>\) Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << Uint(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

### No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn | Rt |
| L  | R  | opcode |

8-bit (opcode == 001)


16-bit (opcode == 011 && size == x0)


32-bit (opcode == 101 && size == 00)


64-bit (opcode == 101 && S == 0 && size == 01)


```plaintext
t = UInt(Rt);
n = UInt(Rn);
m = integer UNKNOWN;
wb = FALSE;
```

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn | Rt |
| L  | R  | opcode |

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wb = FALSE;
8-bit, immediate offset (Rm == 11111 && opcode == 001)


8-bit, register offset (Rm != 11111 && opcode == 001)


16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)


16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)


32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)


32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)


64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)


64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)


```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

**Assembler Symbols**

<VT> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<VT2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<VT3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<VT4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".

For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

For the 32-bit variant: is the element index, encoded in "Q:S".

For the 64-bit variant: is the element index, encoded in "Q".

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
      scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
  integer datasize = if Q == '1' then 128 else 64;
  integer esize = 8 << scale;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADD has no memory ordering semantics.
- STADDL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADD, LDADDA, LDADDAL, LDADDL. This means:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

### Integer

(ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit LDADD alias (size == 10 && R == 0)

STADD <Ws>, [<Xn|SP>]

is equivalent to

LDADD <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 32-bit LDADDL alias (size == 10 & R == 1)

STADDL <Ws>, [<Xn|SP>]

is equivalent to

LDADDL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDADD alias (size == 11 && R == 0)

STADD <Xs>, [<Xn|SP>]

is equivalent to

LDADD <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDADDL alias (size == 11 && R == 1)

STADDL <Xs>, [<Xn|SP>]

is equivalent to

LDADDL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDB has no memory ordering semantics.
- STADDLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB. This means:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

### Integer (ARMv8.1)

<table>
<thead>
<tr>
<th>size</th>
<th>A</th>
<th>opc</th>
<th>Rn</th>
<th>Rs</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

No memory ordering (R == 0)

STADDB <Ws>, [<Xn|SP>]

is equivalent to

LDADDB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STADDLB <Ws>, [<Xn|SP>]

is equivalent to

LDADDLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH has no memory ordering semantics.
- STADDLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Integer (ARMv8.1)

|   |   |   |   |   | R |   |   |   | A |   | Rs |   |   |   |   |   |   | Rn |   |   | Rt |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R | 1  | Rs | 0  | 0  | 0  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |   |   |   |   |   |

No memory ordering (R == 0)

STADDH <Ws>, [<Xn|SP>]

is equivalent to

LDADDH <Ws>, WZR, [Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STADDLH <Ws>, [Xn|SP>]

is equivalent to

LDADDLH <Ws>, WZR, [Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR has no memory ordering semantics.
- STCLRL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL. This means:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

**Integer (ARMv8.1)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |    |

**32-bit LDCLR alias (size == 10 & R == 0)**

STCLR \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDCLR \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

**32-bit LDCLRL alias (size == 10 & R == 1)**

STCLRL \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDCLRL \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

**64-bit LDCLR alias (size == 11 & R == 0)**

STCLR \(<Xs>, [<Xn|SP>]\)

is equivalent to

LDCLR \(<Xs>, XZR, [<Xn|SP>]\)

and is always the preferred disassembly.

**64-bit LDCLRL alias (size == 11 & R == 1)**

STCLRL \(<Xs>, [<Xn|SP>]\)

is equivalent to

LDCLRL \(<Xs>, XZR, [<Xn|SP>]\)

and is always the preferred disassembly.
Assembler Symbols

\(<W_s>\>
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

\(<X_s>\>
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

\(<X_n|SP>\>
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of \(\text{LDCLR}, \text{LDCLRA}, \text{LDCLRAL}, \text{LDCLRL}\) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- **STCLRB** has no memory ordering semantics.
- **STCLRLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB**. This means:

- The encodings in this description are named to match the encodings of **LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB**.
- The description of **LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB** gives the operational pseudocode for this instruction.

### Integer (ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  |

#### No memory ordering (**R == 0**)

**STCLRB <Ws>, [<Xn|SP>]**

is equivalent to

**LDCLRB <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

#### Release (**R == 1**)

**STCLRLB <Ws>, [<Xn|SP>]**

is equivalent to

**LDCLRLB <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRH has no memory ordering semantics.
- STCLRLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH. This means:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Integer
(ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No memory ordering (R == 0)

STCLRH <Ws>, [<Xn|SP>]

is equivalent to

LDCLRH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STCLRLH <Ws>, [<Xn|SP>]

is equivalent to

LDCLRLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEOR has no memory ordering semantics.
- STEORL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDEOR, LDEORA, LDEORAL, LDEORL. This means:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

### Integer

<table>
<thead>
<tr>
<th>Size</th>
<th>A</th>
<th>opc</th>
<th>Rs</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>1 1 1 1 0 0 0 0</td>
<td>R 1</td>
<td>0 0 1 0 0 0</td>
<td>1 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit LDEOR alias (size == 10 && R == 0)

STEOR `<Ws>`, `[<Xn|SP>]`

is equivalent to

LDEOR `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

#### 32-bit LDEORL alias (size == 10 && R == 1)

STEORL `<Ws>`, `[<Xn|SP>]`

is equivalent to

LDEOR `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

#### 64-bit LDEOR alias (size == 11 && R == 0)

STEOR `<Xs>`, `[<Xn|SP>]`

is equivalent to

LDEOR `<Xs>`, XZR, `[<Xn|SP>]`

and is always the preferred disassembly.

#### 64-bit LDEORL alias (size == 11 && R == 1)

STEORL `<Xs>`, `[<Xn|SP>]`

is equivalent to

LDEORL `<Xs>`, XZR, `[<Xn|SP>]`

and is always the preferred disassembly.
Assembler Symbols

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs>  Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORB has no memory ordering semantics.
- STEORLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB. This means:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  |
| size | A | opc | Rn | Rt |

No memory ordering (R == 0)

STEORB <Ws>, [<Xn|SP>]

is equivalent to

LDEORB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STEORLB <Ws>, [<Xn|SP>]

is equivalent to

LDEORLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with
the value held in a register on it, and stores the result back to memory.

- **STEORH** has no memory ordering semantics.
- **STEORLH** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDEORH, LDEORAH, LDEORALH, LDEORLH**. This means:

- The encodings in this description are named to match the encodings of **LDEORH, LDEORAH, LDEORALH, LDEORLH**.
- The description of **LDEORH, LDEORAH, LDEORALH, LDEORLH** gives the operational pseudocode for this instruction.

### Integer

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |

#### No memory ordering (R == 0)

STEORH <Ws>, [Xn|SP]

is equivalent to

LDEORH <Ws>, WZR, [Xn|SP]

and is always the preferred disassembly.

#### Release (R == 1)

STEORLH <Ws>, [Xn|SP]

is equivalent to

LDEORLH <Ws>, WZR, [Xn|SP]

and is always the preferred disassembly.

### Assembler Symbols

- **<Ws>** is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDEORH, LDEORAH, LDEORALH, LDEORLH** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

**Post-index**

(ARMv8.5)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 1 1 0 1 0 0 1 1 0 0 1 | imm9 | 0 | 1 | Xn |
```

| (1) (1) (1) (1) |

**Post-index**

STG [<Xn|SP>], #<simm>

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
boolean zero_data = FALSE;

**Pre-index**

(ARMv8.5)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 1 1 0 0 1 1 0 0 1 | imm9 | 1 | 1 | Xn |
```

| (1) (1) (1) (1) |

**Pre-index**

STG [<Xn|SP>], #<simm>]

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
boolean zero_data = FALSE;

**Signed offset**

(ARMv8.5)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 1 1 0 0 1 0 0 1 | imm9 | 1 | 0 | Xn |
```

| (1) (1) (1) (1) |

**Signed offset**

STG [<Xn|SP>{, #<simm>})

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
boolean zero_data = FALSE;
Assembler Symbols

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```c
bits(64) address;

SetNotTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if zero_data then
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

MemTag[address] = AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset.

### Post-index

(ARMv8.5)

```
0 1 1 0 1 0 0 0 1 0  simm7  Xt2  Xn  Xt
```

- integer n = Uint(Xn);
- integer t = Uint(Xt);
- integer t2 = Uint(Xt2);
- bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
- boolean writeback = TRUE;
- boolean postindex = TRUE;

### Pre-index

(ARMv8.5)

```
0 1 1 0 1 0 0 1 1 0  simm7  Xt2  Xn  Xt
```

- integer n = Uint(Xn);
- integer t = Uint(Xt);
- integer t2 = Uint(Xt2);
- bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
- boolean writeback = TRUE;
- boolean postindex = FALSE;

### Signed offset

(ARMv8.5)

```
0 1 1 0 1 0 0 1 0 0  simm7  Xt2  Xn  Xt
```

- integer n = Uint(Xn);
- integer t = Uint(Xt);
- integer t2 = Uint(Xt2);
- bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
- boolean writeback = FALSE;
- boolean postindex = FALSE;
Assembler Symbols

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<imm> For the post-index and pre-index variant: is the signed immediate offset, in the range -64 to 63, encoded in the "simm7" field.
For the signed offset variant: is the optional signed immediate offset, in the range -64 to 63, defaulting to 0 and encoded in the "simm7" field.

Operation

```
bits(64) address;
bits(64) data1;
bits(64) data2;

SetNotTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data1 = X[t];
data2 = X[t2];

if !postindex then
    address = address + offset;

Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

MemTag[address] = AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Store Tag Vector reads from the second source register an IMPLEMENTATION DEFINED number of Allocation Tags and stores them to the naturally aligned array of 16 allocation tags which includes a tag whose address is the address in the first source register. The Allocation Tag at the address in the first source register is always stored, and the first source register is updated to the address of the first Allocation Tag at an address higher than the original address that was not loaded.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

### Integer (ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Xn | Xt |

### Integer

`STGV <Xt>, [<Xn|SP>]` !

```plaintext
integer t = UInt(Xt);
integer n = UInt(Xn);
```

### Assembler Symbols

- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

### Operation

```plaintext
bits(64) data = X[t];
bits(64) address;

if n == 31 then
    CheckSPLignment();
    address = SP[];
else
    address = X[n];

// address<63:LOG2_TAG_GRANULE+4>:Zeros(LOG2_TAG_GRANULE+4) <= start <= address
// 0 < count <= 16-start<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>
integer count;
(address, count) = ImpDefTagArrayStartAndCount(address);

for i = 0 to count-1
    integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
    bits(4) tag = data<(index*4)+3:index*4>;
    MemTag[address] = tag;
    address = address + TAG_GRANULE;

if n == 31 then
    SP[] = address;
else
    X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

No offset

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | (1)| (1)| (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) |

size | L | Rs | o0 | Rt2 |

32-bit (size == 10)

STLLR <Wt>, [<Xn|SP>@{},#0]

64-bit (size == 11)

STLLR <Xt>, [<Xn|SP>@{},#0]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

case memop of
when MemOp_STORE
  data = X[t];
  Mem[address, dbytes, acctype] = data;
when MemOp_LOAD
  data = Mem[address, dbytes, acctype];
  X[t] = ZeroExtend(data, regsize);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

No offset (ARMv8.1)

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

integer n = Uint(Rn);
integer t = Uint(Rt);
integer t2 = Uint(Rt2); // ignored by load/store single register
integer s = Uint(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << Uint(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;

Assembler Symbols

<Wr> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, dbytes, acctype] = data;
  when MemOp_LOAD
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

**No offset**

(ARMv8.1)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size| 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | (1)| (1)| (1)| (1)| 0  | (1)| (1)| (1)| (1)| Rn  |   |   |   |   |   |   |   |   |   |   |
|     | L  | Rs | o0 | Rt2|

**Assembler Symbols**

<Wr> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);

if n == 31 then
  CheckSPLalignment();
  address = SP[];
else
  address = X[n];

case memop of
  when MemOp STORE
    data = X[t];
    Mem[address, dbytes, acctype] = data;
  when MemOp LOAD
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

32-bit (size == 10)

STLR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STLR <Xt>, [<Xn|SP>{,#0}]

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt () then
    SetNotTagCheckedInstruction(n == 31);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 0  | 0  | 1  | (1) | (1) | (1) | (1) | 1  | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) |

size | L | Rs | o0 | Rt2

No offset

STLRB <Wt>, [<Xn|SP>{1,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType LIMITEDORDERED else AccType ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);
if n == 31 then
    CheckSPAlignment();
else
    address = X[n];

case memop of
    when MemOp STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;
    when MemOp LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00b8f_8c3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

No offset

STLRH <Wt>, [<Xn|SP>{,0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);
if n == 31 then
    CheckSPAlignment();
else
    address = SP[];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLUR

Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release.

For information about memory accesses, see Load/Store addressing modes.

32-bit (size == 10)

```plaintext
STLUR <Wt>, [<Xn|SP>{, #<simm>}
```

64-bit (size == 11)

```plaintext
STLUR <Xt>, [<Xn|SP>{, #<simm>]
```

Assembler Symbols

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
  if opc<0> == '1' then UNDEFINED;
else
  // sign-extending load
  memop = MemOp_LOAD;
  if size == '10' && opc<0> == '1' then UNDEFINED;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS      wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN         wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF           UNDEFINED;
        when Constraint_NOP             EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE            rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN         rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF           UNDEFINED;
        when Constraint_NOP             EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[ ];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[ ] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

- **Unscaled offset**

  STLURB `<Wt>`, `[<Xn>SP], #<simm>`

  ```
  boolean wback = FALSE;
  boolean postindex = FALSE;
  integer scale = UInt(size);
  bits(64) offset = SignExtend(imm9, 64);
  ```

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' & & opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```
if `HaveMTEExt()` then
    boolean `is_load_store` = `memop` IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(`is_load_store` && `n` == 31 && !`wback`);

bits(64) `address`;
bits(datasize) `data`;

boolean `wb_unknown` = FALSE;
boolean `rt_unknown` = FALSE;

if `memop` == MemOp_LOAD && `wback` && `n` == `t` && `n` != 31 then
    `c` = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert `c` IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case `c` of
        when Constraint_WBSUPPRESS `wback` = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN `wb_unknown` = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    end case;

if `memop` == MemOp_STORE && `wback` && `n` == `t` && `n` != 31 then
    `c` = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert `c` IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case `c` of
        when Constraint_NONE `rt_unknown` = FALSE;  // value stored is original value
        when Constraint_UNKNOWN `rt_unknown` = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    end case;

if `n` == 31 then
    if `memop` != MemOp_PREFETCH then CheckSPAlignment();
    `address` = SP[];
else
    `address` = X[n];

if ! `postindex` then
    `address` = `address` + `offset`;

case `memop` of
    when MemOp_STORE
        if `rt_unknown` then
            `data` = bits(datasize) UNKNOWN;
        else
            `data` = X[t];
            Mem[`address`, datasize DIV 8, acctype] = `data`;
    when MemOp_LOAD
        `data` = Mem[`address`, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(`data`, regsize);
        else
            X[t] = ZeroExtend(`data`, regsize);
    when MemOp_PREFETCH
        Prefetch(`address`, t<4:0>);
end case;

if `wback` then
    if `wb_unknown` then
        `address` = bits(64) UNKNOWN;
    elseif `postindex` then
        `address` = `address` + `offset`;
    if `n` == 31 then
        SP[] = `address`;
    else
        X[n] = `address`;
end if;
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLURH

Store-Release Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release.

For information about memory accesses, see Load/Store addressing modes.

### Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`: Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainsUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainsUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;  // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

For information about memory accesses see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | sz | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | Rs | 1  | Rt2| Rn | 00 |

32-bit (sz == 0)

STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

| integer n = UInt(Rn); |
| integer t = UInt(Rt); |
| integer t2 = UInt(Rt2); // ignored by load/store single register |
| integer s = UInt(Rs);   // ignored by all loads and store-release |
| AccType acctype = if o0 == '1' then AccType_ORDERED_ATOMIC else AccType_ATOMIC; |
| boolean pair = TRUE; |
| MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE; |
| integer elsize = 32 << UInt(sz); |
| integer regsize = if elsize == 64 then 64 else 32; |
| integer datasize = if pair then elsize * 2 else elsize; |

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXP.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rt" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.
If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN (Constraint UNKNOWN, Constraint UNDEF, Constraint NOP);
  case c of
    when Constraint UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint UNDEF UNDEFINED;
    when Constraint NOP EndOfInstruction();
  if memop == MemOp STORE then
    if s == t || (pair && s == t2) then
      Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
      assert c IN (Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP);
      case c of
        when Constraint UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint NONE rt_unknown = FALSE; // store original value
        when Constraint UNDEF UNDEFINED;
        when Constraint NOP EndOfInstruction();
      if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN (Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP);
        case c of
          when Constraint UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
          when Constraint NONE rn_unknown = FALSE; // address is original base
          when Constraint UNDEF UNDEFINED;
          when Constraint NOP EndOfInstruction();
      if n == 31 then
        CheckSPAlignment();
        address = SP[];
      elsif rn_unknown then
        address = bits(64) UNKNOWN;
      else
        address = X[n];
      case memop of
        when MemOp STORE
          if rt_unknown then
            data = bits(datasize) UNKNOWN;
          elseif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
          else
            data = X[t];
          bit status = '1';
          // Check whether the Exclusives monitors are set to include the
          // physical memory locations corresponding to virtual address
          // range [address, address+dbytes-1].
          if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);
          when MemOp LOAD
            // Tell the Exclusives monitors to record a sequence of one or more atomic
            // memory reads from virtual address range [address, address+dbytes-1].
            // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
  if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
  elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, acctype];
    if BigEndian() then
      X[t] = data<datasize-1:elsize>;
      X[t2] = data<elsize-1:0>;
    else
      X[t] = data<elsize-1:0>;
      X[t2] = data<datasize-1:elsize>;
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
      X[t] = Mem[address + 0, 8, acctype];
      X[t2] = Mem[address + 8, 8, acctype];
    else
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtend(data, regsize);
  end
else
  data = Mem[address, dbytes, acctype];
  X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

32-bit (size == 10)

STLXR <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STLXR <Ws>, <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXR.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xx> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wx> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:
• Memory is not updated.
• <Ws> is not updated.
Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
• If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
• Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.
If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = Construnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint UNDEF UNDEFINED;
    when Constraint NOP EndOfInstruction();
  end case;
if memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = Construnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP};
    case c of
      when Constraint UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
      when Constraint NONE rt_unknown = FALSE; // store original value
      when Constraint UNDEF UNDEFINED;
      when Constraint NOP EndOfInstruction();
    end case;
  end if;
if s == n && n != 31 then
  Constraint c = Construnpredictable(Unpredictable_BASEOVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
    when Constraint NONE rn_unknown = FALSE; // address is original base
    when Constraint UNDEF UNDEFINED;
    when Constraint NOP EndOfInstruction();
  end case;
if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];
end case;

memop of
  when MemOp STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    elseif pair then
      bits(datasize DIV 2) el1 = X[t];
      bits(datasize DIV 2) el2 = X[t2];
      data = if BigEndian() then el1 : el2 else el2 : el1;
    else
      data = X[t];
    end if;
    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
      // This atomic write will be rejected if it does not refer
      // to the same physical locations after address translation.
      Mem[address, dbytes, acctype] = data;
      status = ExclusiveMonitorsStatus();
      X[s] = ZeroExtend(status, 32);
    end if;
  when MemOp LOAD
    // Tell the Exclusives monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
  if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
  elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, acctype];
    if BigEndian() then
      X[t] = data<datasize-1:elsize>;
      X[t2] = data<elsize-1:0>;
    else
      X[t] = data<elsize-1:0>;
      X[t2] = data<datasize-1:elsize>;
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
    end then
      X[t] = Mem[address + 0, 8, acctype];
      X[t2] = Mem[address + 8, 8, acctype];
  end then
  else
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(data, regsize);
end then

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

No offset

STLXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRB.

Assembler Symbols

<Ws>       Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0         If the operation updates memory.
1         If the operation fails to update memory.

<Wt>       Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>    Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts
If a synchronous Data Abort exception is generated by the execution of this instruction:
  • Memory is not updated.
  • <Ws> is not updated.
If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);
if memop == MemOp LOAD && pair && t == t2 then
    Constraint c = ConstrainsUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if memop == MemOp STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainsUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NOP EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainsUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_NONE rn_unknown = FALSE; // address is original base
        when Constraint_UNDEF rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_NOP EndOfInstruction();
if n == 31 then
    CheckSPAlignment();
    address = SP[ ];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
case memop of
    when MemOp STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];
    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = CheckExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);
    when MemOp LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
        end;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
        end;
    end;
end;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Rs</th>
<th></th>
<th></th>
<th></th>
<th>Rn</th>
<th></th>
<th></th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

No offset

STLXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly STLXRH.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Aborts and alignment**

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
bits(64) address;
bis(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE;  // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE;  // store UNKNOWN value
      when Constraint_NONE rt_unknown = FALSE;  // store original value
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rn_unknown = TRUE;  // address is UNKNOWN
      when Constraint_NONE rn_unknown = FALSE;  // address is original base
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

case memop of
  when MemOp STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    elsif pair then
      bits(datasize DIV 2) el1 = X[t];
      bits(datasize DIV 2) el2 = X[t2];
      data = if BigEndian() then el1 : el2 else el2 : el1;
    else
      data = X[t];
    
    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
      // This atomic write will be rejected if it does not refer
      // to the same physical locations after address translation.
      Mem[address, dbytes, acctype] = data;
      status = ExclusiveMonitorsStatus();
      X[s] = ZeroExtend(status, 32);

  when MemOp LOAD
    // Tell the Exclusives monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
        end
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        end
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
    end
end

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point Non-temporal pair.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>opc</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm7</td>
<td>Rt2</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]  

64-bit (opc == 01)

STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]  

128-bit (opc == 10)

STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]  

boolean wback = FALSE;  
boolean postindex = FALSE;

Assembler Symbols

<Dt1>  
Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt2>  
Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Qt1>  
Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt2>  
Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<St1>  
Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<St2>  
Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Xn|SP>  
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rt" field.

<imm>  
For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer t2 = UInt(Rt2);  
AccType acctype = AccType_VECSTREAM;  
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;  
if opc == '11' then UNDEFINED;  
integer scale = 2 + UInt(opc);  
integer datasize = 8 << scale;  
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
Operation

```c
CheckFPAdvSIMEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == MemOp LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0 , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;
    when MemOp LOAD
        data1 = Mem[address + 0 , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t] = data1;
        V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see Load/Store addressing modes. For information about Non-temporal pair instructions, see Load/Store Non-temporal pair.

32-bit (opc == 00)

STNP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]

64-bit (opc == 10)

STNP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]

boolean wback = FALSE;
boolean postindex = FALSE;

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint.UNKNOWN, Constraint.UNDEF, Constraint.NOP};
    case c of
        when Constraint.UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint.UNDEF UNDEFINED;
        when Constraint.NOP EndOfInstruction();
    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];
    if !postindex then
        address = address + offset;
    case memop of
        when MemOp_STORE
            if rt_unknown && t == n then
                data1 = bits(datasize) UNKNOWN;
            else
                data1 = X[t];
            if rt_unknown && t2 == n then
                data2 = bits(datasize) UNKNOWN;
            else
                data2 = X[t2];
            Mem[address + 0 , dbytes, acctype] = data1;
            Mem[address + dbytes, dbytes, acctype] = data2;
        when MemOp_LOAD
            data1 = Mem[address + 0 , dbytes, acctype];
            data2 = Mem[address + dbytes, dbytes, acctype];
            if rt_unknown then
                data1 = bits(datasize) UNKNOWN;
                data2 = bits(datasize) UNKNOWN;
                X[t] = data1;
                X[t2] = data2;
            end
            if wback then
                if postindex then
                    address = address + offset;
                if n == 31 then
                    SP[] = address;
                else
                    X[n] = address;
    Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: **Post-index**, **Pre-index** and **Signed offset**

**Post-index**

<table>
<thead>
<tr>
<th>opc</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

```java
boolean wback = TRUE;
boolean postindex = TRUE;
```

**Pre-index**

<table>
<thead>
<tr>
<th>opc</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>!

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>!

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>!

```java
boolean wback = TRUE;
boolean postindex = FALSE;
```

**Signed offset**

<table>
<thead>
<tr>
<th>opc</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

12-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>!

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>!

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>!
32-bit (opc == 00)

\[
\text{STP } \langle St1 \rangle, \langle St2 \rangle, [\langle Xn|SP \rangle\{, \#<imm>\}]
\]

64-bit (opc == 01)

\[
\text{STP } \langle Dt1 \rangle, \langle Dt2 \rangle, [\langle Xn|SP \rangle\{, \#<imm>\}]
\]

128-bit (opc == 10)

\[
\text{STP } \langle Qt1 \rangle, \langle Qt2 \rangle, [\langle Xn|SP \rangle\{, \#<imm>\}]
\]

boolean \( wback = \text{FALSE} \);
boolean \( \text{postindex} = \text{FALSE} \);

Assembler Symbols

\(<Dt1>\) Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<Dt2>\) Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
\(<Qt1>\) Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<Qt2>\) Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
\(<St1>\) Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<St2>\) Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\(<imm>\) For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.
For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.
For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.
For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.
For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

integer \( n = \text{UInt}(Rn); \)
integer \( t = \text{UInt}(Rt); \)
integer \( t2 = \text{UInt}(Rt2); \)
\( \text{AccType acctype = AccType_VEC}; \)
\( \text{MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE}; \)
if opc == '11' then UNDEFINED;
integer scale = 2 + \text{UInt}(opc);
integer datasize = 8 << scale;
bits(64) \( \text{offset = LSL} \text{SignExtend}(\text{imm7}, 64), \text{scale}); \)
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstranUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0 , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;
    when MemOp_LOAD
        data1 = Mem[address + 0 , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t] = data1;
        V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00ber8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**STP**

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset.

### Post-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td><strong>imm7</strong></td>
<td><strong>Rt2</strong></td>
<td><strong>Rn</strong></td>
<td><strong>Rt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (opc == 00)**

```plaintext
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

**64-bit (opc == 10)**

```plaintext
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```plaintext
boolean wback  = TRUE;
boolean postindex = TRUE;
```

### Pre-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td><strong>imm7</strong></td>
<td><strong>Rt2</strong></td>
<td><strong>Rn</strong></td>
<td><strong>Rt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (opc == 00)**

```plaintext
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>!
```

**64-bit (opc == 10)**

```plaintext
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>!
```

```plaintext
boolean wback  = TRUE;
boolean postindex = TRUE;
```

### Signed offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td><strong>imm7</strong></td>
<td><strong>Rt2</strong></td>
<td><strong>Rn</strong></td>
<td><strong>Rt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (opc == 00)**

```plaintext
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

**64-bit (opc == 10)**

```plaintext
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```plaintext
boolean wback  = FALSE;
boolean postindex = FALSE;
```
For information about the constrained unpredictable behavior of this instruction, see "Architectural Constraints on Unpredictable Behaviors," and particularly STP.

**Assembler Symbols**

- `<Wt1>` is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Wt2>` is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xt1>` is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt2>` is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`
  - For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as `<imm>/4`.
  - For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as `<imm>/4`.
  - For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as `<imm>/8`.
  - For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as `<imm>/8`.

**Shared Decode**

```plaintext
integer n = Uint(Rn);
integer t = Uint(Rt);
integer t2 = Uint(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + Uint(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

  boolean wb_unknown = FALSE;
  if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
      when Constraint WBSUPPRESS wback = FALSE; // writeback is suppressed
      when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
      when Constraint UNDEF UNDEFINED;
      when Constraint NOP EndOfInstruction();
  if memop == MemOp STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
      when Constraint NONE rt_unknown = FALSE; // value stored is pre-writeback
      when Constraint UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
      when Constraint UNDEF UNDEFINED;
      when Constraint NOP EndOfInstruction();
  if memop == MemOp LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
      when Constraint UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
      when Constraint UNDEF UNDEFINED;
      when Constraint NOP EndOfInstruction();
  if n == 31 then
    CheckSPAlignment();
    address = SP[];
  else
    address = X[n];
    if ! postindex then
      address = address + offset;
    case memop of
      when MemOp STORE
        if rt_unknown && t == n then
          data1 = bits(datasize) UNKNOWN;
        else
          data1 = X[t];
        if rt_unknown && t2 == n then
          data2 = bits(datasize) UNKNOWN;
        else
          data2 = X[t2];
          Mem[address + 0 , dbytes, acctype] = data1;
          Mem[address + dbytes, dbytes, acctype] = data2;
      when MemOp LOAD
        data1 = Mem[address + 0 , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
          data1 = bits(datasize) UNKNOWN;
          data2 = bits(datasize) UNKNOWN;
        if signed then
          X[t] = SignExtend(data1, 64);
          X[t2] = SignExtend(data2, 64);
        else
          STP
X[t] = data1;
X[t2] = data2;

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

### Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| size| 1   | 1   | 1   | 1   | 0   | 0   | x   | 0   | 0   | imm9| 0   | 1   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

8-bit (size == 00 && opc == 00)

STR <Bt>, [<Xn|SP>], #<simm>

16-bit (size == 01 && opc == 00)

STR <Ht>, [<Xn|SP>], #<simm>

32-bit (size == 10 && opc == 00)

STR <St>, [<Xn|SP>], #<simm>

64-bit (size == 11 && opc == 00)

STR <Dt>, [<Xn|SP>], #<simm>

128-bit (size == 00 && opc == 10)

STR <Qt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bias(64) offset = SignExtend(imm9, 64);

### Pre-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| size| 1   | 1   | 1   | 1   | 0   | 0   | x   | 0   | 0   | imm9| 1   | 1   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

opc
8-bit ($size == 00$ && $opc == 00$)

\[
\text{STR} \ <Bt>, \ [<Xn|SP>], \ #\text{simm}]
\]

16-bit ($size == 01$ && $opc == 00$)

\[
\text{STR} \ <Ht>, \ [<Xn|SP>], \ #\text{simm}]
\]

32-bit ($size == 10$ && $opc == 00$)

\[
\text{STR} \ <St>, \ [<Xn|SP>], \ #\text{simm}]
\]

64-bit ($size == 11$ && $opc == 00$)

\[
\text{STR} \ <Dt>, \ [<Xn|SP>], \ #\text{simm}]
\]

128-bit ($size == 00$ && $opc == 10$)

\[
\text{STR} \ <Qt>, \ [<Xn|SP>], \ #\text{simm}]
\]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt($opc<1>:size$);
if scale > 4 then UNDEFINED;
bounds(64) offset = SignExtend($imm9, 64$);

**Unsigned offset**

| size | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|      | 1  | 1  | 1  | 1  | 0  | 1  | x  | 0  | imm12 | Rn | Rt |

opc

8-bit ($size == 00$ && $opc == 00$)

\[
\text{STR} \ <Bt>, \ [<Xn|SP>], \ #\text{pimm}]
\]

16-bit ($size == 01$ && $opc == 00$)

\[
\text{STR} \ <Ht>, \ [<Xn|SP>], \ #\text{pimm}]
\]

32-bit ($size == 10$ && $opc == 00$)

\[
\text{STR} \ <St>, \ [<Xn|SP>], \ #\text{pimm}]
\]

64-bit ($size == 11$ && $opc == 00$)

\[
\text{STR} \ <Dt>, \ [<Xn|SP>], \ #\text{pimm}]
\]

128-bit ($size == 00$ && $opc == 10$)

\[
\text{STR} \ <Qt>, \ [<Xn|SP>], \ #\text{pimm}]
\]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt($opc<1>:size$);
if scale > 4 then UNDEFINED;
bounds(64) offset = LSL(ZeroExtend($imm12, 64$), scale);
Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

```
integer n = Uint(Rn);
integer t = Uint(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```
Operation

```c
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    CheckFPAdvSIMDEnabled64();
    bits(64) address;
    bits(datasize) data;

    if n == 31 then
        CheckSPAAlignment();
        address = SP[];
    else
        address = X[n];

    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp_STORE
            data = V[t];
            Mem[address, datasize DIV 8, acctype] = data;
        when MemOp LOAD
            data = Mem[address, datasize DIV 8, acctype];
            V[t] = data;

        if wback then
            if postindex then
                address = address + offset;
            if n == 31 then
                SP[] = address;
            else
                X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | imm9| 0   | 1   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

size opc

32-bit (size == 10)

STR <Wt>, [<Xn|SP>], #<simm>

64-bit (size == 11)

STR <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bites(64) offset = SignExtend(imm9, 64);

Pre-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | imm9| 1   | 1   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

size opc

32-bit (size == 10)

STR <Wt>, [<Xn|SP>], #<simm>

64-bit (size == 11)

STR <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bites(64) offset = SignExtend(imm9, 64);

Unsigned offset

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 1   | 1   | 0   | 0   | 0   | 1   | 0   | imm12|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

size opc
32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>{, #<pimm>}
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>{, #<pimm>}
```

```plaintext
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>` For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as `<pimm>/4`. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as `<pimm>/8`.

**Shared Decode**

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

STR (immediate)
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>size</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>opc</td>
</tr>
</tbody>
</table>
```

8-fsreg,STR-8-fsreg (size == 00 && opc == 00 && option != 011)

```
STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]`
```

8-fsreg,STR-8-fsreg (size == 00 && opc == 00 && option == 011)

```
STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-fsreg,STR-16-fsreg (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-fsreg,STR-32-fsreg (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-fsreg,STR-64-fsreg (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-fsreg,STR-128-fsreg (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
Integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- `<Bt>` Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Dt>` Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Ht>` Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Qt>` Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<St>` Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rt" field.
- `<Wm>` When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>` When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>` For the 8-bit variant: is the index extend specifier, encoded in "option".
For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when `<amount>` is omitted. encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when `<extend>` is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is the index shift amount, optional only when `<extend>` is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when `<extend>` is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>

For the 128-bit variant: is the index shift amount, optional only when `<extend>` is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#4</td>
</tr>
</tbody>
</table>

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
Integer datasize = 8 << scale;
```
bits(64) offset = \texttt{ExtendReg}(m, extend\_type, shift);
if \texttt{HaveMTEExt}() then
    boolean is\_load\_store = memop IN \{\texttt{MemOp\_STORE}, \texttt{MemOp\_LOAD}\};
    SetNotTagCheckedInstruction(is\_load\_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if ! postindex then
    address = address + offset;

\textbf{case memop of}
\begin{itemize}
    \item \texttt{MemOp\_STORE} \\
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;
    \item \texttt{MemOp\_LOAD} \\
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;
\end{itemize}
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

\textbf{Operational information}

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit
doubleword to the calculated address, from a register. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset
register value. The offset can be optionally shifted and extended.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
| 1  | x  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | Rm |   | option | S  | 1  | 0  | Rn |   | Rt |   |
| size | opc |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
```

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, (Wm<M|M>), <extend> {<amount}>]
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, (Wm<M|M>), <extend> {<amount}>]
```

```java
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;  // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>` When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>` When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted.
  encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it
  defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

- `<amount>` For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it
  defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN rb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE;  // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
        else
            address = X[n];
        if ! postindex then
            address = address + offset;
        case memop of
            when MemOp_STORE
                if rt_unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = X[t];
                    Mem[address, datasize DIV 8, acctype] = data;
            when MemOp_LOAD
                data = Mem[address, datasize DIV 8, acctype];
                if signed then
                    X[t] = SignExtend(data, regsize);
                else
                    X[t] = ZeroExtend(data, regsize);
            when MemOp_PREFETCH
                Prefetch(address, t<4:0>);
    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

**Post-index**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm9</td>
<td></td>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size   opc

```java
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

**Pre-index**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm9</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size   opc

```java
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

**Unsigned offset**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>imm12</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size   opc

```java
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STRB (immediate).

**Assembler Symbols**

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

### Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType accType = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
   // store or zero-extending load
   memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
   regsize = if size == '1l' then 64 else 32;
   signed = FALSE;
else
   if size == 'll' then UNDEFINED;
   else
      // sign-extending load
      memop = MemOp_LOAD;
      if size == '10' && opc<0> == '1' then UNDEFINED;
      regsize = if opc<0> == '1' then 32 else 64;
      signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    bits(64) address;
    bits(datasize) data;

    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if memop == MemOp LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
    
    if memop == MemOp STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
        when MemOp LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

Extended register (option != 011)

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  
```

boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
if option<1> == '0' then UNDEFINED; // sub-word index  

```
   ExtendType extend_type = DecodeRegExtend(option);  
   integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>` When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>` When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>` Is the index extend specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- `<amount>` Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType accType = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if ! postindex then
    address = address + offset;

switch memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

**Post-index**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|---------------------------|---------------------------|
| 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| size | opc | imm9 | Rn | Rt |

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

**Pre-index**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|---------------------------|---------------------------|
| 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| size | opc | imm9 | Rn | Rt |

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

**Unsigned offset**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|---------------------------|---------------------------|
| 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| size | opc | imm12 | Rn | Rt |

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STRH (immediate)*.

**Assembler Symbols**

- **<Wt>** Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    bits(64) address;
    bits(datasize) data;

    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if memop == MemOp LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
        when MemOp LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | Rm | option | S   | 1   | 0   | Rn  | Rt  |
```

**32-bit**

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>)], <extend> {<amount>}
```

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

**Assembler Symbols**

- **<Wt>**
  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

- **<Xn|SP>**
  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- **<Wm>**
  When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

- **<Xm>**
  When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

- **<extend>**
  Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. Encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- **<amount>**
  Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
Operation

```plaintext
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

STRH (register)
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSET, STSE TL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSET has no memory ordering semantics.
- STSE TL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSET, LDSE TA, LDSE TAL, LDSE TL. This means:

- The encodings in this description are named to match the encodings of LDSET, LDSE TA, LDSE TAL, LDSE TL.
- The description of LDSET, LDSE TA, LDSE TAL, LDSE TL gives the operational pseudocode for this instruction.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1 | 1 | 1 | 0 | 0 | 0 | 0 | R | 1 | Rs | 0 | 0 | 1 | 1 | 0 | Rn | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

32-bit LDSET alias (size == 10 && R == 0)

STSET <Ws>, [<Xn|SP>]  
is equivalent to  
LDSET <Ws>, WZR, [<Xn|SP>]  
and is always the preferred disassembly.

32-bit LDSE TL alias (size == 10 && R == 1)

STSE TL <Ws>, [<Xn|SP>]  
is equivalent to  
LDSE TL <Ws>, WZR, [<Xn|SP>]  
and is always the preferred disassembly.

64-bit LDSET alias (size == 11 && R == 0)

STSET <Xs>, [<Xn|SP>]  
is equivalent to  
LDSET <Xs>, XZR, [<Xn|SP>]  
and is always the preferred disassembly.

64-bit LDSE TL alias (size == 11 && R == 1)

STSE TL <Xs>, [<Xn|SP>]  
is equivalent to  
LDSE TL <Xs>, XZR, [<Xn|SP>]  
and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- **STSETB** has no memory ordering semantics.
- **STSETLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSETB, LDSETAB, LDSETALB, LDSETLB**. This means:

- The encodings in this description are named to match the encodings of **LDSETB, LDSETAB, LDSETALB, LDSETLB**.
- The description of **LDSETB, LDSETAB, LDSETALB, LDSETLB** gives the operational pseudocode for this instruction.

### Integer

**(ARMv8.1)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |

**No memory ordering (R == 0)**

**STSETB <Ws>, [<Xn|SP>]**

is equivalent to

**LDSETB <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

**Release (R == 1)**

**STSETLB <Ws>, [<Xn|SP>]**

is equivalent to

**LDSETLB <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDSETB, LDSETAB, LDSETALB, LDSETLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH has no memory ordering semantics.
- STSETLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH. This means:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  |

No memory ordering (R == 0)

STSETH <Ws>, [<Xn|SP>]

is equivalent to

LDSETH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSETLH <Ws>, [<Xn|SP>]

is equivalent to

LDSETLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws>           Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>        Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAX has no memory ordering semantics.
- STSMAXL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL. This means:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

### Integer (ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit LDSMAX alias (size == 10 && R == 0)

STSMAX $<Ws>$, $[<Xn|SP>]$

is equivalent to

LDSMAX $<Ws>$, WZR, $[<Xn|SP>]$

and is always the preferred disassembly.

#### 32-bit LDSMAXL alias (size == 10 && R == 1)

STSMAXL $<Ws>$, $[<Xn|SP>]$

is equivalent to

LDSMAXL $<Ws>$, WZR, $[<Xn|SP>]$

and is always the preferred disassembly.

#### 64-bit LDSMAX alias (size == 11 && R == 0)

STSMAX $<Xs>$, $[<Xn|SP>]$

is equivalent to

LDSMAX $<Xs>$, XZR, $[<Xn|SP>]$

and is always the preferred disassembly.

#### 64-bit LDSMAXL alias (size == 11 && R == 1)

STSMAXL $<Xs>$, $[<Xn|SP>]$

is equivalent to

LDSMAXL $<Xs>$, XZR, $[<Xn|SP>]$

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STSMAXB, STSMAXLB**

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- **STSMAXB** has no memory ordering semantics.
- **STSMAXLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB**. This means:

- The encodings in this description are named to match the encodings of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB**.
- The description of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB** gives the operational pseudocode for this instruction.

### Integer

**(ARMv8.1)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | R | 1 | Rs  | 0 | 1 | 0 | 0 | 0 | Rn | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | R | 1 |

#### No memory ordering (R == 0)

**STSMAXB** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDSMAXB** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

#### Release (R == 1)

**STSMAXLB** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDSMAXLB** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

### Assembler Symbols

- `<Ws>` is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB** gives the operational pseudocode for this instruction.

### Operational Information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXH has no memory ordering semantics.
- STSMAXLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH. This means:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

### Integer

(ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>A</td>
<td>opc</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### No memory ordering (R == 0)

STSMAX <Ws>, [<Xn|SP>]

is equivalent to

LDSMAX <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### Release (R == 1)

STSMAXLH <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

---

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMIN has no memory ordering semantics.
- STSMINL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL. This means:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |
| size | A  | opc |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

32-bit LDSMIN alias (size == 10 && R == 0)

STSMIN <Ws>, [Xn|SP]

is equivalent to

LDSMIN <Ws>, WZR, [Xn|SP]

and is always the preferred disassembly.

32-bit LDSMINL alias (size == 10 && R == 1)

STSMINL <Ws>, [Xn|SP]

is equivalent to

LDSMINL <Ws>, WZR, [Xn|SP]

and is always the preferred disassembly.

64-bit LDSMIN alias (size == 11 && R == 0)

STSMIN <Xs>, [Xn|SP]

is equivalent to

LDSMIN <Xs>, XZR, [Xn|SP]

and is always the preferred disassembly.

64-bit LDSMINL alias (size == 11 && R == 1)

STSMINL <Xs>, [Xn|SP]

is equivalent to

LDSMINL <Xs>, XZR, [Xn|SP]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- **STSMINB** has no memory ordering semantics.
- **STSMINLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB**. This means:

- The encodings in this description are named to match the encodings of **LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB**.
- The description of **LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB** gives the operational pseudocode for this instruction.

**Integer**

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |

**No memory ordering (R == 0)**

**STSMINB** `<Ws>`, `[<Xn|SP>]`

is equivalent to

**LDSMINB** `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

**Release (R == 1)**

**STSMINLB** `<Ws>`, `[<Xn|SP>]`

is equivalent to

**LDSMINLB** `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of **LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINH has no memory ordering semantics.
- STSMINLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH. This means:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

### Integer

(ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**No memory ordering (R == 0)**

STSMINH `<Ws>`, `<Xn|SP>`

is equivalent to

LDSMINH `<Ws>`, WZR, `<Xn|SP>`

and is always the preferred disassembly.

**Release (R == 1)**

STSMINLH `<Ws>`, `<Xn|SP>`

is equivalent to

LDSMINLH `<Ws>`, WZR, `<Xn|SP>`

and is always the preferred disassembly.

### Assembler Symbols

- `<Ws>` is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

32-bit (size == 10)

```
STTR <Wt>, [<Xn|SP>{, #<simm>}
```

64-bit (size == 11)

```
STTR <Xt>, [<Xn|SP>{, #<simm>}
```

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

- **<Wt>** Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xt>** Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<simm>** Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = HaveEL(EL2) && HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11');

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

  bits(64) address;
  bits(datasize) data;
  boolean wb_unknown = FALSE;
  boolean rt_unknown = FALSE;

  if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
      when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();

  if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
      when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();

  if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
  else
    address = X[n];

  if ! postindex then
    address = address + offset;

  case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
      data = Mem[address, datasize DIV 8, acctype];
      if signed then
        X[t] = SignExtend(data, regsize);
      else
        X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
      Prefetch(address, t<4:0>);

  if wback then
    if wb_unknown then
      address = bits(64) UNKNOWN;
    elsif postindex then
      address = address + offset;
    if n == 31 then
      SP[] = address;
    else
      X[n] = address;
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

Unscaled offset

STTRB \langle Wt \rangle, [\langle Xn|SP \rangle\{, \#<simm>\}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

\langle Wt \rangle  
Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\langle Xn|SP \rangle
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\langle simm \rangle
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = HaveEL(EL2) && HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    UNDEFINED;
  else // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);
    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.<E2H,TGE> is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

### Unscaled offset

STTRH <Wt>, [<Xn|SP>{, #<simm}>]

- boolean wback = FALSE;
- boolean postindex = FALSE;
- integer scale = UInt(size);
- bits(64) offset = SignExtend(imm9, 64);

#### Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`: Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

#### Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = HaveEL(EL2) && HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN \{MemOp STORE, MemOp LOAD\};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp LOAD && wback && n == t && n != 31 then
    c = ConstruUnpredictable(\Unpredictable\WBOVERLAPLD\);
    assert c IN \{Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP\};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when Constraint NOP EndOfInstruction();

if memop == MemOp STORE && wback && n == t && n != 31 then
    c = ConstruUnpredictable(\Unpredictable\WBOVERLAPST\);
    assert c IN \{Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP\};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
        
    when MemOp LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAX has no memory ordering semantics.
- STUMAXL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL. This means:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 1  | 1  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |

32-bit LDUMAX alias (size == 10 && R == 0)

STUMAX \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDUMAX \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

32-bit LDUMAXL alias (size == 10 && R == 1)

STUMAXL \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDUMAXL \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

64-bit LDUMAX alias (size == 11 && R == 0)

STUMAX \(<Xs>, [<Xn|SP>]\)

is equivalent to

LDUMAX \(<Xs>, XZR, [<Xn|SP>]\)

and is always the preferred disassembly.

64-bit LDUMAXL alias (size == 11 && R == 1)

STUMAXL \(<Xs>, [<Xn|SP>]\)

is equivalent to

LDUMAXL \(<Xs>, XZR, [<Xn|SP>]\)

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- **STUMAXB** has no memory ordering semantics.
- **STUMAXLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB**. This means:

- The encodings in this description are named to match the encodings of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB**.
- The description of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB** gives the operational pseudocode for this instruction.

### Integer

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | R | 1  | Rs | 0  | 1  | 1  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1 |

**No memory ordering (R == 0)**

\[
\text{STUMAXB } \langle Ws \rangle, \ [\langle Xn|SP \rangle]
\]

is equivalent to

\[
\text{LDUMAXB } \langle Ws \rangle, \ WZR, \ [\langle Xn|SP \rangle]
\]

and is always the preferred disassembly.

**Release (R == 1)**

\[
\text{STUMAXLB } \langle Ws \rangle, \ [\langle Xn|SP \rangle]
\]

is equivalent to

\[
\text{LDUMAXLB } \langle Ws \rangle, \ WZR, \ [\langle Xn|SP \rangle]
\]

and is always the preferred disassembly.

### Assembler Symbols

- **<Ws>** is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH has no memory ordering semantics.
- STUMAXLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH. This means:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

**Integer**
(ARMv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**size**  A  **opc**  Rt

No memory ordering (R == 0)

STUMAXH <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STUMAXLH <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STUMIN, STUMINL**

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- **STUMIN** has no memory ordering semantics.
- **STUMINL** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDUMIN, LDUMINA, LDUMINAL, LDUMINL**. This means:

- The encodings in this description are named to match the encodings of **LDUMIN, LDUMINA, LDUMINAL, LDUMINL**.
- The description of **LDUMIN, LDUMINA, LDUMINAL, LDUMINL** gives the operational pseudocode for this instruction.

### Integer (ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 1  | 1  | 1  | 0  | Rn | 1  | 1  | 1  | 1  | 1  | |

#### 32-bit LDUMIN alias (size == 10 && R == 0)

**STUMIN** `<Ws>`, `[<Xn|SP>]`

is equivalent to

**LDUMIN** `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

#### 32-bit LDUMINL alias (size == 10 && R == 1)

**STUMINL** `<Ws>`, `[<Xn|SP>]`

is equivalent to

**LDUMINL** `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

#### 64-bit LDUMIN alias (size == 11 && R == 0)

**STUMIN** `<Xs>`, `[<Xn|SP>]`

is equivalent to

**LDUMIN** `<Xs>`, XZR, `[<Xn|SP>]`

and is always the preferred disassembly.

#### 64-bit LDUMINL alias (size == 11 && R == 1)

**STUMINL** `<Xs>`, `[<Xn|SP>]`

is equivalent to

**LDUMINL** `<Xs>`, XZR, `[<Xn|SP>]`

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- **STUMINB** has no memory ordering semantics.
- **STUMINLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB**. This means:

- The encodings in this description are named to match the encodings of **LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB**.
- The description of **LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB** gives the operational pseudocode for this instruction.

### Integer (ARMv8.1)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | R  | 1  | Rs  | 0  | 1  | 1  | 1  | 0  | 0  | Rn  | 1  | 1  | 1  | 1  | 1  |
| A    | opc |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rt   |     |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

#### No memory ordering (R == 0)

**STUMINB** <<Ws>, [<Xn|SP>]]

is equivalent to

**LDUMINB** <<Ws>, WZR, [<Xn|SP>]]

and is always the preferred disassembly.

#### Release (R == 1)

**STUMINLB** <<Ws>, [<Xn|SP>]]

is equivalent to

**LDUMINLB** <<Ws>, WZR, [<Xn|SP>]]

and is always the preferred disassembly.

### Assembler Symbols

- **<Ws>** is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINH has no memory ordering semantics.
- STUMINLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH. This means:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

### Integer

(ARMv8.1)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R | 1  | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |   |   |   |   |   |

#### No memory ordering (R == 0)

STUMINH <Ws>, [<Xn|SP>]

is equivalent to

LDUMINH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### Release (R == 1)

STUMINLH <Ws>, [<Xn|SP>]

is equivalent to

LDUMINLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Assembler Symbols

- `<Ws>` is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>0</th>
<th>imm9</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
</tr>
</tbody>
</table>

8-bit (size == 00 && opc == 00)

STUR <Bt>, [<Xn|SP>{, #<simm>}

16-bit (size == 01 && opc == 00)

STUR <Ht>, [<Xn|SP>{, #<simm>}

32-bit (size == 10 && opc == 00)

STUR <St>, [<Xn|SP>{, #<simm>}

64-bit (size == 11 && opc == 00)

STUR <Dt>, [<Xn|SP>{, #<simm>}

128-bit (size == 00 && opc == 10)

STUR <Qt>, [<Xn|SP>{, #<simm>}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);  

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
Operation

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    data = V[t];
    Mem[address, datasize DIV 8, acctype] = data;

  when MemOp_LOAD
    data = Mem[address, datasize DIV 8, acctype];
    V[t] = data;

if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STUR**

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see Load/Store addressing modes.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| x | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | imm9 | 0 | 0 | Rn | Rt |

**32-bit (size == 10)**

STUR <Wt>, [<Xn|SP>{, #<simm>}

**64-bit (size == 11)**

STUR <Xt>, [<Xn|SP>{, #<simm>}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

- **<Wt>**
  - Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

- **<Xt>**
  - Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

- **<Xn|SP>**
  - Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- **<simm>**
  - Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
  if opc<0> == '1' then UNDEFINED;
else
  // sign-extending load
  memop = MemOp_LOAD;
  if size == '10' && opc<0> == '1' then UNDEFINED;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;
integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF       UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF       UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | imm9| 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| size| opc|

Unscaled offset

STURB <Wt>, [<Xn|SP>{, #<simm}>]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
  integer datasize = 8 << scale;
if \texttt{HaveMTEExt()} then
    boolean \texttt{is\_load\_store} = memop \texttt{IN (MemOp\_STORE, MemOp\_LOAD)};
    \texttt{SetNotTagCheckedInstruction(is\_load\_store \&\& n == 31 \&\& !wback)};

\texttt{bits(64) address;}
\texttt{bits(datasize) data;}

boolean \texttt{wb\_unknown} = FALSE;
boolean \texttt{rt\_unknown} = FALSE;

if memop == \texttt{MemOp\_LOAD} \&\& wback \&\& n == t \&\& n != 31 then
    c = \texttt{ConstrainUnpredictable(Unpredictable\_WBOVERLAPLD)};
    \texttt{assert c \texttt{IN (Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP)}};
    \texttt{case c of}
    \texttt{case c of}
    \texttt{when Constraint\_WBSUPPRESS wback = FALSE; \textwidth {writeback is suppressed}}
    \texttt{when Constraint\_UNKNOWN wb\_unknown = TRUE; \textwidth {writeback is UNKNOWN}}
    \texttt{when Constraint\_UNDEF \textwidth {UNDEFINED}};
    \texttt{when Constraint\_NOP \textwidth {EndOfInstruction()}};

if memop == \texttt{MemOp\_STORE} \&\& wback \&\& n == t \&\& n != 31 then
    c = \texttt{ConstrainUnpredictable(Unpredictable\_WBOVERLAPST)};
    \texttt{assert c \texttt{IN (Constraint\_NONE, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP)}};
    \texttt{case c of}
    \texttt{when Constraint\_NONE rt\_unknown = FALSE; \textwidth {value stored is original value}}
    \texttt{when Constraint\_UNKNOWN rt\_unknown = TRUE; \textwidth {value stored is UNKNOWN}}
    \texttt{when Constraint\_UNDEF \textwidth {UNDEFINED}};
    \texttt{when Constraint\_NOP \textwidth {EndOfInstruction()}};

if n == 31 then
    if memop != \texttt{MemOp\_PREFETCH} then \texttt{CheckSPAlignment();}
    \texttt{address = \texttt{SP}[];}  
    else
    \texttt{address = X[n];}

if ! postindex then
    \texttt{address = address + offset;}

\texttt{case memop of}
\texttt{when MemOp\_STORE}
    if rt\_unknown then
        \texttt{data = bits(datasize) UNKNOWN;}
    else
        \texttt{data = X[t];}
        \texttt{Mem[address, datasize DIV 8, acctype] = data;}
\texttt{when MemOp\_LOAD}
    \texttt{data = Mem[address, datasize DIV 8, acctype];}
    \texttt{if signed then}
        \texttt{X[t] = SignExtend(data, regsize);}  
    \texttt{else}
        \texttt{X[t] = ZeroExtend(data, regsize);}  
\texttt{when MemOp\_PREFETCH}
    \texttt{Prefetch(address, t<4:0>);}

if wback then
    if wb\_unknown then
        \texttt{address = bits(64) UNKNOWN;}
    \texttt{elseif postindex then}
        \texttt{address = address + offset;}
    if n == 31 then
        \texttt{SP[]} = address;
    else
        \texttt{X[n] = address;}

\texttt{STURB}
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see Load/Store addressing modes.

Unscaled offset

STURH <Wt>, [<Xn|SP>{, #<simm}>]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = if size == '11' then 64 else 32;
  signed = FALSE;
else
  if size == '11' then
    memop = MemOp_PREFETCH;
    if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN \{MemOp STORE, MemOp LOAD\};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

    bits(64) address;
    bits(datasize) data;

    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if memop == MemOp LOAD && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        assert c IN \{Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP\};
        case c of
            when Constraint_WBSUPPRESS wbback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
        assert c IN \{Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP\};
        case c of
            when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE;   // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[ ];
    else
        address = X[n];

    if ! postindex then
        address = address + offset;

    case memop of
        when MemOp STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, datasize DIV 8, acctype] = data;
        when MemOp LOAD
            data = Mem[address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[ ] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses see Load/Store addressing modes.

### Assembler Symbols

- `<Ws>`: Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
  - 0: If the operation updates memory.
  - 1: If the operation fails to update memory.
- `<Xt1>`: Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt2>`: Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Wt1>`: Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Wt2>`: Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:
- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN (Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP);
  case c of
    when Constraint_UNKNOWN     rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF        UNDEFINED;
    when Constraint_NOP          EndOfInstruction();

if memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN (Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP);
    case c of
      when Constraint_UNKNOWN     rt_unknown = TRUE; // store UNKNOWN value
      when Constraint_NONE         rt_unknown = FALSE; // store original value
      when Constraint_UNDEF        UNDEFINED;
      when Constraint_NOP          EndOfInstruction();
  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN (Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP);
    case c of
      when Constraint_UNKNOWN     rn_unknown = TRUE; // address is UNKNOWN
      when Constraint_NONE         rn_unknown = FALSE; // address is original base
      when Constraint_UNDEF        UNDEFINED;
      when Constraint_NOP          EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

  case memop of
    when MemOp STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      elseif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian() then el1 : el2 else el2 : el1;
      else
        data = X[t];
      end;
      bit status = '1';
      // Check whether the Exclusives monitors are set to include the
      // physical memory locations corresponding to virtual address
      // range [address, address+dbytes-1].
      if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);
    when MemOp LOAD
      // Tell the Exclusives monitors to record a sequence of one or more atomic
      // memory reads from virtual address range [address, address+dbytes-1].
      // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        $X[t]$ = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            $X[t]$ = data<datasize-1:elsize>;
            $X[t2]$ = data<elsize-1:0>;
        else
            $X[t]$ = data<elsize-1:0>;
            $X[t2]$ = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            $X[t]$ = Mem[address + 0, 8, acctype];
            $X[t2]$ = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            $X[t]$ = ZeroExtend(data, regsize);
    end

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

32-bit (size == 10)

STXR <Ws>, <Wt>, [<Xn|SP>,<#0>]

64-bit (size == 11)

STXR <Ws>, <Xt>, [<Xn|SP>,<#0>]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);  // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXR.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rt" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:
- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.
If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
bits(64) address;
brightness(data_size) data;
constant integer dbytes = data_size DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExe() then
    bovemblem is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store & & n == 31);
if memop == MemOp LOAD & & pair & & t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if memop == MemOp STORE then
    if s == t | (pair & & s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
    if s == n & & n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
            when Constraint_NONE rn_unknown = FALSE; // address is original base
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    elsif rn_unknown then
        address = bits(64) UNKNOWN;
    else
        address = X[n];
    case memop of
        when MemOp STORE
            if rt_unknown then
                data = bits(data_size) UNKNOWN;
            elseif pair then
                bits(data_size DIV 2) el1 = X[t];
                bits(data_size DIV 2) el2 = X[t2];
                data = if BigEndian() then el1 : el2 else el2 : el1;
            else
                data = X[t];
            bit status = '1';
            // Check whether the Exclusives monitors are set to include the
            // physical memory locations corresponding to virtual address
            // range [address, address+dbytes-1].
            if AArch64.ExclusiveMonitorsPass(address, dbytes) then
                // This atomic write will be rejected if it does not refer
                // to the same physical locations after address translation.
                Mem[address, dbytes, acctype] = data;
                status = ExclusiveMonitorsStatus();
                X[s] = ZeroExtend(status, 32);
            else
                // Tell the Exclusives monitors to record a sequence of one or more atomic
                // memory reads from virtual address range [address, address+dbytes-1].
                // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
  if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
  elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, acctype];
    if BigEndian() then
      X[t] = data<datasize-1:elsize>;
      X[t2] = data<elsize-1:0>;
    else
      X[t] = data<elsize-1:0>;
      X[t2] = data<datasize-1:elsize>;
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
      X[t] = Mem[address + 0, 8, acctype];
      X[t2] = Mem[address + 8, 8, acctype];
    else
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtend(data, regsize);
  end

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic.

For information about memory accesses see Load/Store addressing modes.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29 | 28| 27| 26 | 25| 24 | 23 | 22| 21| 20| 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | 0  |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

No offset

STXRB $<Ws>$, $<Wt>$, [$<Xn|SP>$,0]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType accctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXRB.

Assembler Symbols

$<Ws>$ Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

$<Wt>$ Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

$<Xn|SP>$ Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts
If a synchronous Data Abort exception is generated by the execution of this instruction:
• Memory is not updated.
• $<W>$ is not updated.
If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);
if memop == MemOp LOAD && pair && t == t2 then
  Constraint c = ConstrainUnpredictable(Unexpected LDPOVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN      rt_unknown = TRUE;  // result is UNKNOWN
    when Constraint UNDEF        UNDEFINED;
    when Constraint NOP          EndOfInstruction();
if memop == MemOp STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unexpected DATAOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP};
    case c of
      when Constraint UNKNOWN     rt_unknown = TRUE;  // store UNKNOWN value
      when Constraint NONE        rt_unknown = FALSE;  // store original value
      when Constraint UNDEF       UNDEFINED;
      when Constraint NOP         EndOfInstruction();
  if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unexpected BASEOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint NONE, Constraint UNDEF, Constraint NOP};
    case c of
      when Constraint UNKNOWN     rn_unknown = TRUE;  // address is UNKNOWN
      when Constraint NONE        rn_unknown = FALSE;  // address is original base
      when Constraint UNDEF       UNDEFINED;
      when Constraint NOP         EndOfInstruction();
if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

case memop of
  when MemOp STORE
    if rt_unknown then
      data = bits(datasize) UNKNOWN;
    elsif pair then
      bits(datasize DIV 2) el1 = X[t];
      bits(datasize DIV 2) el2 = X[t2];
      data = if BigEndian() then el1 : el2 else el2 : el1;
    else
      data = X[t];
    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
      // This atomic write will be rejected if it does not refer
      // to the same physical locations after address translation.
      Mem[address, dbytes, acctype] = data;
      status = ExclusiveMonitorsStatus();
      X[s] = ZeroExtend(status, 32);
  when MemOp LOAD
    // Tell the Exclusives monitors to record a sequence of one or more atomic
    // memory reads from virtual address range [address, address+dbytes-1].
    // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
  if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    \text{X}[t] = \text{bits(datasize)} \text{UNKNOWN};
  elsif elseize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = \text{Mem}[address, dbytes, acctype];
    if BigEndian() then
      \text{X}[t] = \text{data}\langle\text{datasize-1:elseize}\rangle;
      \text{X}[t2] = \text{data}\langle\text{elseize-1:0}\rangle;
    else
      \text{X}[t] = \text{data}\langle\text{elseize-1:0}\rangle;
      \text{X}[t2] = \text{data}\langle\text{datasize-1:elseize}\rangle;
    end
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != \text{Align}(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
    \text{X}[t] = \text{Mem}[address + 0, 8, acctype];
    \text{X}[t2] = \text{Mem}[address + 8, 8, acctype];
  else
    data = \text{Mem}[address, dbytes, acctype];
    \text{X}[t] = \text{ZeroExtend}(data, regsize);
  end

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic.

For information about memory accesses see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>o0</th>
<th>Rt2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

No offset

STXRH \(<Ws\), \(<Wt\), [\(<Xn|SP>|\{,#0}\)]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED_ATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer datasize = if pair then elsize * 2 else elsize;

Assembler Symbols

\(<Ws>\)

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

\(<Wt>\)

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\)

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<Ws>\) is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp LOAD && pair && t == t2 then
    Constraint c = ConstrainsUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint.UNKNOWN, Constraint.UNDEF, Constraint.NOP};
    case c of
        when Constraint.UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint.UNDEF UNDEFINED;
        when Constraint.NOP EndOfInstruction();

if memop == MemOp STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainsUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint.UNKNOWN, Constraint.NONE, Constraint.UNDEF, Constraint.NOP};
        case c of
            when Constraint.UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
            when Constraint.NONE rt_unknown = FALSE; // store original value
            when Constraint.UNDEF UNDEFINED;
            when Constraint.NOP EndOfInstruction();

if s == n && n != 31 then
    Constraint c = ConstrainsUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint.UNKNOWN, Constraint.NONE, Constraint.UNDEF, Constraint.NOP};
    case c of
        when Constraint.UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint.NONE rn_unknown = FALSE; // address is original base
        when Constraint.UNDEF UNDEFINED;
        when Constraint.NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
elif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
  if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
  elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, acctype];
    if BigEndian() then
      X[t] = data<datasize-1:elsize>;
      X[t2] = data<elsize-1:0>;
    else
      X[t] = data<elsize-1:0>;
      X[t2] = data<datasize-1:elsize>;
  else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
      iswrite = FALSE;
      secondstage = FALSE;
      AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
      X[t] = Mem[address + 0, 8, acctype];
      X[t2] = Mem[address + 8, 8, acctype];
    else
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtend(data, regsize);
  end

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STZ2G

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index

(ARMv8.5)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | imm9 | 0 | 1 | Xn | (1) | (1) | (1) | (1) | (1) |

Post-index

STZ2G [<Xn|SP>], #<simm>

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
boolean zero_data = TRUE;

Pre-index

(ARMv8.5)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | imm9 | 1 | 1 | Xn | (1) | (1) | (1) | (1) | (1) |

Pre-index

STZ2G [<Xn|SP>, #<simm>]

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
boolean zero_data = TRUE;

Signed offset

(ARMv8.5)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | imm9 | 1 | 0 | Xn | (1) | (1) | (1) | (1) | (1) |

Signed offset

STZ2G [<Xn|SP>, #<simm>]

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
boolean zero_data = TRUE;
Assembler Symbols

<Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<simm>

Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```c
bits(64) address;
bits(4) tag;

SetNotTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if zero_data then
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8*TAG_GRANULE);
    Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(8*TAG_GRANULE);

tag = AllocationTagFromAddress(address);

MemTag[address] = tag;
MemTag[address+TAG_GRANULE] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**STZG**

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

### Post-index
**(ARMv8.5)**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | imm9| 0  | 1  | Xn | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) |
```

```
stzg [<Xn|SP>], #<simm>
```

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
boolean zero_data = TRUE;

### Pre-index
**(ARMv8.5)**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | imm9| 1  | 1  | Xn | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) |
```

```
stzg [<Xn|SP>, #<simm>]
```

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
boolean zero_data = TRUE;

### Signed offset
**(ARMv8.5)**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | imm9| 1  | 0  | Xn | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) |
```

```
stzg [<Xn|SP>], #<simm>]
```

integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
boolean zero_data = TRUE;
Assembler Symbols

<Xn|SP>
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<simm>
Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```c
bits(64) address;
SetNotTagCheckedInstruction(TRUE);
if n == 31 then
    CheckSPAignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
if zero_data then
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
    MemTag[address] = AllocationTagFromAddress(address);
if writeback then
    if postindex then
        address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SUB (extended register)**

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
<th>option</th>
<th>imm3</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

### 32-bit (sf == 0)

\[ \text{SUB} \ <Wd|WSP>, \ <Wn|WSP>, \ <Wm>{, \ <extend> \ (#<amount>)} \]

### 64-bit (sf == 1)

\[ \text{SUB} \ <Xd|SP>, \ <Xn|SP>, \ <R><m>{, \ <extend> \ (#<amount>)} \]

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;

**Assembler Symbols**

- `<Wd|WSP>`: Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn|WSP>`: Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd|SP>`: Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn|SP>`: Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<R>`: Is a width specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

- `<m>`: Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- `<extend>`: For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UTXX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTX</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

**SUB (extended register)**
If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;
if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';
(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUB (immediate)**

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>! = 1x</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

### 64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

```plaintext
d = UInt(Rd);
n = UInt(Rn);
datasize = if sf == '1' then 64 else 32;
sub_op = (op == '1');
setflags = (S == '1');
imm = case shift of
  when '00' => ZeroExtend(imm12, datasize);
  when '01' => ZeroExtend(imm12 : Zeros(12), datasize);
  when '10' => SEE "ADDG, SUBG";
  when '11' => ReservedValue();
```

**Assembler Symbols**

- `<Wd|WSP>`: Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn|WSP>`: Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd|SP>`: Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn|SP>`: Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>`: Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

<table>
<thead>
<tr>
<th>shift&lt;0&gt;</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
  operand2 = NOT(operand2);
  carry_in = '1';
else
  carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
  PSTATE.<N,Z,C,V> = nzcv;
if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUB (shifted register)**

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias **NEG (shifted register)**.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 1 0 0 1 0 1 1 shift 0 Rm imm6 Rn Rd
```

**32-bit (sf == 0)**

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

**64-bit (sf == 1)**

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- `<amount>` For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG (shifted register)</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

---

**SUB (shifted register)**
Operation

\begin{verbatim}
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
\end{verbatim}

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUB (vector)**

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

|   |   |   |   |   |   |   | size |   |   |   | Rm |   |   |   |   | Rd |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24 | 23| 22| 21| 20| 19| 18| 17| 16| 15 | 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
| U |   |   |   |   |   |   | size|   |   |   |    |   |   |   |   |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

### Vector

|   |   |   |   |   |   |   | size |   |   |   | Rm |   |   |   |   |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0   | size|   |   |    |   |   |   |   |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

### Assembler Symbols

- **<V>** is a width specifier, encoded in "size":
  - size | <V> |
  - 0x   | RESERVED |
  - 10   | RESERVED |
  - 11   | D |

- **<d>** is the number of the SIMD&FP destination register, in the "Rd" field.
- **<n>** is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- **<m>** is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Vd>** is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** is an arrangement specifier, encoded in "size:Q":

### Example

Scalar

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUBG**

Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

### Integer (ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | uimm6 | (0) | (0) | uimm4 | Xn | Xd |

### Assembler Symbols

- `<Xd|SP>`: Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
- `<Xn|SP>`: Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<uimm6>`: Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
- `<uimm4>`: Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

### Operation

```python
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(4) tag_offset = uimm4;
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
boolean ADD = FALSE;

bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AllocationTagAccessIsEnabled() then
    rtag = ChooseNonExcludedTag(start_tag, tag_offset, exclude);
else
    rtag = '0000';

if ADD then
    (result, -) = AddWithCarry(operand1, offset, '0');
else
    (result, -) = AddWithCarry(operand1, NOT(offset), '1');
result = AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00ber8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SUBHN, SUBHN2**

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see **RSUBHN**.

The SUBHN instruction writes the vector to the lower half of the destination register and clears the upper half; while the SUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

---

### Three registers, not all the same type

**SUBHN(2) <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

---

**Assembler Symbols**

- **Q**
  
  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

- **<Vd>**
  
  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Tb>**
  
  Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>01</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
</tr>
</tbody>
</table>

- **<Vn>**
  
  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- **<Ta>**
  
  Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vm>**
  
  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
  element1 = Elem[operand1, e, 2*esize];
  element2 = Elem[operand2, e, 2*esize];
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  sum = sum + round_const;
  Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

**Integer (ARMv8.5)**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**Assemble Symbol**

\[
\text{SUBP } <Xd>, <Xn|SP>, <Xm|SP>
\]

- \( \text{integer } d = \text{UInt}(Xd) \)
- \( \text{integer } n = \text{UInt}(Xn) \)
- \( \text{integer } m = \text{UInt}(Xm) \)
- \( \text{boolean setflags} = \text{FALSE} \)

**Operation**

\[
\begin{align*}
\text{bits(64) operand1} & = \text{if } n == 31 \text{ then SP[] else } X[n]; \\
\text{bits(64) operand2} & = \text{if } m == 31 \text{ then SP[] else } X[m]; \\
\text{operand1} & = \text{SignExtend}(\text{operand1}<55:0>, 64); \\
\text{operand2} & = \text{SignExtend}(\text{operand2}<55:0>, 64); \\
\text{bits(64) result}; \\
\text{bits(4) nzcv}; \\
\text{operand2} & = \text{NOT(operand2)}; \\
(\text{result}, \text{nzcv}) & = \text{AddWithCarry}(\text{operand1}, \text{operand2}, '1'); \\
\text{if } \text{setflags} \text{ then} \\
\text{PSTATE.<N,Z,C,V>} & = \text{nzcv}; \\
X[d] & = \text{result};
\end{align*}
\]
SUBPS

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias CMPP.

Integer

(ARMv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |  Xm |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    | Xn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Integer

SUBPS <Xd>, <Xn|SP>, <Xm|SP>

integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
boolean setflags = TRUE;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPP</td>
<td>S == '1' &amp;&amp; Xd == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) operand2 = if m == 31 then SP[] else X[m];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
bits(4) nzcv;
operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
    X[d] = result;
**SUBS (extended register)**

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias **CMP (extended register)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
<th>option</th>
<th>imm3</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

64-bit (sf == 1)

SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;

**Assembler Symbols**

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '111111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in “option”:
If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount>
Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP (extended register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(datasize) result;
bits(datasize) operando1 = if n == 31 then SP[] else X[n];
bits(datasize) operando2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operando2 = NOT(operando2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operando1, operando2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUBS (immediate)**

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **CMP (immediate)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>op</th>
<th>S</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm12</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

**64-bit (sf == 1)**

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

integer \(d\) = UInt(Rd);
integer \(n\) = UInt(Rn);
integer \(datasize\) = if sf == '1' then 64 else 32;
boolean \(sub\)_op = (op == '1');
boolean \(set\)_flags = (S == '1');
bits(datasize) \(imm\);

```
case shift of
    when '00' \(imm\) = ZeroExtend(imm12, datasize);
    when '01' \(imm\) = ZeroExtend(imm12 : Zeros(12), datasize);
    when '10' SEE "ADDG, SUBG";
    when '11' ReservedValue();
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

<table>
<thead>
<tr>
<th>shift&lt;0&gt;</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP (immediate)</strong></td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUBS (shifted register)**

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases [CMP (shifted register)](https://example.com) and [NEGS](https://example.com).

### 32-bit (sf == 0)

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

**Assembler Symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>`: Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>`: For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
<tr>
<td>NEGS</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

|
Operation

```c
bits(datasize) result;
bits(datasize) operand1 = \texttt{X}[n];
bits(datasize) operand2 = \texttt{ShiftReg}(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;
if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = \texttt{AddWithCarry}(operand1, operand2, carry_in);
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;
\texttt{X}[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SUQADD

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit $FPSR.QC$ is set. Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Scalar**

SUQADD $<V><d>$, $<V><n>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

**Vector**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Vector**

SUQADD $<Vd>.<T>$, $<Vn>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

**Assembler Symbols**

- `<V>` is a width specifier, encoded in "size":
  - | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
  - | 00 | B |
  - | 01 | H |
  - | 10 | S |
  - | 11 | D |

- `<d>` is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<n>` is the number of the SIMD&FP source register, encoded in the "Rn" field.
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “size-Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
    V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SVC**

Supervisor Call causes an exception to be taken to EL1.

On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in $ESR_{ELx}$, using the EC value 0x15, and the value of the immediate argument.

```
+----------------+-----------------+
|  31 30 29 28 27 26 25 24 23 22 21 20 19 | 18 17 16 15 14 13 12 11 10  9  8  7  6 |  5  4  3  2  1  0 |
| 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 0 1 1 0 1 0 | 0 0

System

SVC #<imm>

```
bits(16) imm = imm16;
```

**Assembler Symbols**

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

**Operation**

```
AArch64.CallSupervisor(imm);
```
**SWP, SWPA, SWPAL, SWPL**

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, SWPA and SWPAL load from memory with acquire semantics.
- SWPL and SWPAL store to memory with release semantics.
- SWP has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

**ARMv8.1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 1  | 0  | 0  | 0  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**size**
32-bit SWP (size == 10 && A == 0 && R == 0)

SWP <Ws>, <Wt>, [<Xn|SP>]

32-bit SWPA (size == 10 && A == 1 && R == 0)

SWPA <Ws>, <Wt>, [<Xn|SP>]

32-bit SWPAL (size == 10 && A == 1 && R == 1)

SWPAL <Ws>, <Wt>, [<Xn|SP>]

32-bit SWPL (size == 10 && A == 0 && R == 1)

SWPL <Ws>, <Wt>, [<Xn|SP>]

64-bit SWP (size == 11 && A == 0 && R == 0)

SWP <Xs>, <Xt>, [<Xn|SP>]

64-bit SWPA (size == 11 && A == 1 && R == 0)

SWPA <Xs>, <Xt>, [<Xn|SP>]

64-bit SWPAL (size == 11 && A == 1 && R == 1)

SWPAL <Xs>, <Xt>, [<Xn|SP>]

64-bit SWPL (size == 11 && A == 0 && R == 1)

SWPL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

```c
bits(64) address;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
Mem[address, datasize DIV 8, stacctype] = X[s];

X[t] = ZeroExtend(data, regsize);
```
SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

Integer
(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rn |   |   |   |   |   |

size

SWPB (A == 1 && R == 0)

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

SWPAB (A == 1 && R == 1)

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

SWPB (A == 0 && R == 0)

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

SWPLB (A == 0 && R == 1)

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

```python
if !HaveAtomicExt() then UNDEFINED;

t = UInt(Rt);
n = UInt(Rn);
s = UInt(Rs);

datasize = 8 << UInt(size);
regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

Assembler Symbols

- `<Ws>` Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- `<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

```plaintext
bits(64) address;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
Mem[address, datasize DIV 8, stacctype] = X[s];

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00be68_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has no memory ordering requirements.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

### Integer (ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs  | 1  | 0  | 0  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

#### SWPAH (A == 1 && R == 0)

SWPAH <Ws>, <Wt>, [<Xn|SP>]

#### SWPALH (A == 1 && R == 1)

SWPALH <Ws>, <Wt>, [<Xn|SP>]

#### SWPH (A == 0 && R == 0)

SWPH <Ws>, <Wt>, [<Xn|SP>]

#### SWPLH (A == 0 && R == 1)

SWPLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- **<Wt>** Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

```c
bits(64) address;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
Mem[address, datasize DIV 8, stacctype] = X[s];

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| sf | 0 0 1 0 0 1 0 | N | 0 0 0 0 0 0 | 0 0 0 1 1 1 | Rn | Rd |
| opc | immr | imms |

32-bit (sf == 0 && N == 0)

SXTB <Wd>, <Wn>

is equivalent to

SBFM <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

SXTB <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #7

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

• The encodings in this description are named to match the encodings of SBFM.
• The description of SBFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>immr</th>
<th>imms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 && N == 0)

SXTH <Wd>, <Wn>

is equivalent to

SBFM <Wd>, <Wn>, #0, #15

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

SXTH <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #15

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
SXTL, SXTL2

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SXTL instruction extracts the source vector from the lower half of the source register, while the SXTL2 instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of SSHLL, SSHLL2. This means:

- The encodings in this description are named to match the encodings of SSHLL, SSHLL2.
- The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.

### Vector

SXTL(2) <Vd>, <Ta>, <Vn>, <Tb>

is equivalent to

SSHLL(2) <Vd>, <Ta>, <Vn>, <Tb>, #0

and is the preferred disassembly when BitCount(immh) == 1.

### Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in &quot;Q&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;Vd&gt;</th>
<th>Is the name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</th>
</tr>
</thead>
</table>
| <Ta> | Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>lxxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;Vn&gt;</th>
<th>Is the name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</th>
</tr>
</thead>
</table>
| <Tb> | Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>lxxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### Operation

The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | Rn |    |    |    |    |    |    |    |    |    |    |

64-bit

SXTW <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #31

and is always the preferred disassembly.

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
System instruction. For more information, see Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions for the encodings of System instructions.

This instruction is used by the aliases AT, CFP, CPP, DC, DVP, IC, and TLBI.

### Assembler Symbols

<op1> is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>CRn == '0111' &amp;&amp; CRm == '100x' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_AT</td>
</tr>
<tr>
<td>CFP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '100'</td>
</tr>
<tr>
<td>CPP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '111'</td>
</tr>
<tr>
<td>DC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_DC</td>
</tr>
<tr>
<td>DVP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '101'</td>
</tr>
<tr>
<td>IC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_IC</td>
</tr>
<tr>
<td>TLBI</td>
<td>CRn == '1000' &amp;&amp; SysOp(op1,'1000',CRm,op2) == Sys_TLBI</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
if has_result then
    X[t] = AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys.crm, sys_op2);
else
    AArch64.SysInstr(sys_op0, sys_op1, sys_crn, sys.crm, sys_op2, X[t]);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00b987_rc3; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**System instruction with result.** For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | op1 | CRn | CRm | op2 | Rt | L |

**System**

SYSL `<Xt>`, `<op1>`, `<Cn>`, `<Cm>`, `<op2>`

```plaintext
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer sys_op0 = 1;
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
boolean has_result = (L == '1');
```

**Assembler Symbols**

- `<Xt>`  
  Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

- `<op1>`  
  Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

- `<Cn>`  
  Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

- `<Cm>`  
  Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

- `<op2>`  
  Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

**Operation**

```plaintext
if has_result then
    X[t] = AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, syscrm, sys_op2);
else
    AArch64.SysInstr(sys_op0, sys_op1, sys_crn, syscrm, sys_op2, X[t]);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>len</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two register table (len == 01)

\texttt{TBL <Vd>.<Ta>, \{ <Vn>.16B, <Vn+1>.16B \}, <Vm>.<Ta>}

Three register table (len == 10)

\texttt{TBL <Vd>.<Ta>, \{ <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B \}, <Vm>.<Ta>}

Four register table (len == 11)

\texttt{TBL <Vd>.<Ta>, \{ <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B \}, <Vm>.<Ta>}

Single register table (len == 00)

\texttt{TBL <Vd>.<Ta>, \{ <Vn>.16B \}, <Vm>.<Ta>}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
\end{verbatim}

Assembler Symbols

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<Ta>} Is an arrangement specifier, encoded in “Q”:

\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{Q} & \texttt{<Ta>} \\
\hline
0 & 8B \\
1 & 16B \\
\hline
\end{tabular}
\end{center}

\texttt{<Vn>} For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

\texttt{<Vn+1>} Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

\texttt{<Vn+2>} Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

\texttt{<Vn+3>} Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

\texttt{<Vm>} Is the name of the SIMD&FP index register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs - 1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

<table>
<thead>
<tr>
<th>b5</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>b40</th>
<th>imm14</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

14-bit signed PC-relative branch offset

TBNZ <R><t>, #<imm>, <label>

```plaintext
integer t = UInt(Rt);
integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bit bit_val = op;
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

- `<R>` Is a width specifier, encoded in “b5”:
  - b5
  - 0: W
  - 1: X

  In assembler source code an ‘X’ specifier is always permitted, but a ‘W’ specifier is only permitted when the bit number is less than 32.

- `<t>` Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

- `<imm>` Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

- `<label>` Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```plaintext
bits(datasize) operand = X[t];
if operand<bit_pos> == bit_val then
  BranchTo(PC[] + offset, BranchType_DIR);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | Rm | 0  | len | 1  | 0  | 0  | Rn | 0  | Rd |

**op**

### Two register table (len == 01)

TBX `<Vd>..<Ta>`, { `<Vn>.16B, <Vn+1>.16B` }, `<Vm>..<Ta>`

### Three register table (len == 10)

TBX `<Vd>..<Ta>`, { `<Vn>.16B, <Vn+1>.16B, <Vn+2>.16B` }, `<Vm>..<Ta>`

### Four register table (len == 11)

TBX `<Vd>..<Ta>`, { `<Vn+.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B` }, `<Vm>..<Ta>`

### Single register table (len == 00)

TBX `<Vd>..<Ta>`, { `<Vn>.16B` }, `<Vm>..<Ta>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

### Assembler Symbols

- `<Vd>` - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ta>` - Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;Ta&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vn>` - For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.
  
  For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

- `<Vn+1>` - Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

- `<Vn+2>` - Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

- `<Vn+3>` - Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

- `<Vm>` - Is the name of the SIMD&FP index register, encoded in the "Rm" field.
Operation

```plaintext
CheckFPAdvSIMDEnabled64();

bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs - 1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| b5 | 0  | 1  | 1  | 0  | 1  | 0  | 1  | b40 |

14-bit signed PC-relative branch offset

TBZ <R><t>, #<imm>, <label>

integer t = UInt(Rt);
integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bit bit_val = op;
bits(64) offset = SignExtend(imm14:'00', 64);

Assembler Symbols

<R> Is a width specifier, encoded in “b5”:

<table>
<thead>
<tr>
<th>b5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

bits(datasize) operand = X[t];
if operand<bit_pos> == bit_val then
   BranchTo(PC[] + offset, BranchType_DIR);

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
TLBI

TLB Invalidate operation. For more information, see .

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

System

TLBI <tlbi_op>{, <Xt>}

is equivalent to

SYS #<op1>, C8, <Cm>, #<op2>{, <Xt>}

and is the preferred disassembly when SysOp(op1,'1000',CRm,op2) == Sys_TLBI.

Assembler Symbols

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<tlbi_op> Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in “op1:CRm:op2”:
<table>
<thead>
<tr>
<th>op1</th>
<th>CRm</th>
<th>op2</th>
<th>&lt;tlbi_op&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>000</td>
<td>VMALLE1OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>001</td>
<td>VAE1OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>ASIDE1OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>011</td>
<td>VAAE1OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>101</td>
<td>VALE1OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>111</td>
<td>VAALE1OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>001</td>
<td>RVAE1IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>011</td>
<td>RVAAE1IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>101</td>
<td>RVALE1IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>111</td>
<td>RVAALE1IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>000</td>
<td>VMALLE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>001</td>
<td>VAE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>010</td>
<td>ASIDE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>011</td>
<td>VAAE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>101</td>
<td>VALE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>111</td>
<td>VAALE1IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>001</td>
<td>IPAS2E1IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>010</td>
<td>RIPAS2E1IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>100</td>
<td>001</td>
<td>000</td>
<td>ALLE2OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>100</td>
<td>001</td>
<td>001</td>
<td>VAE2OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>100</td>
<td>001</td>
<td>100</td>
<td>ALLE1OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>100</td>
<td>001</td>
<td>101</td>
<td>VALE2OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>100</td>
<td>001</td>
<td>110</td>
<td>VMALLS12E1OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>100</td>
<td>010</td>
<td>001</td>
<td>RVAE2IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>100</td>
<td>010</td>
<td>101</td>
<td>RVALE2IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>100</td>
<td>011</td>
<td>000</td>
<td>ALLE2IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>011</td>
<td>001</td>
<td>VAE2IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>011</td>
<td>100</td>
<td>ALLE1IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>011</td>
<td>101</td>
<td>VALE2IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>011</td>
<td>110</td>
<td>VMALLS12E1</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>110</td>
<td>000</td>
<td>000</td>
<td>ALLE3OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
</tbody>
</table>

TLBI
<table>
<thead>
<tr>
<th>op1</th>
<th>CRm</th>
<th>op2</th>
<th>&lt;tlbi_op&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>0001</td>
<td>001</td>
<td>VAE3OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>101</td>
<td>VALE3OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>110</td>
<td>0010</td>
<td>001</td>
<td>RVAE3IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>110</td>
<td>0010</td>
<td>101</td>
<td>RVALE3IS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>000</td>
<td>ALLE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>001</td>
<td>VAE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>101</td>
<td>VALE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0101</td>
<td>001</td>
<td>RVAE3OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>110</td>
<td>0101</td>
<td>101</td>
<td>RVALE3OS</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>110</td>
<td>0110</td>
<td>001</td>
<td>RVAE3</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>110</td>
<td>0110</td>
<td>101</td>
<td>RVALE3</td>
<td>ARMv8.4-TLBI</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>000</td>
<td>ALLE3</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>001</td>
<td>VAE3</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>101</td>
<td>VALE3</td>
<td>-</td>
</tr>
</tbody>
</table>

<Xi> Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the "Rt" field.

**Operation**

The description of SYS gives the operational pseudocode for this instruction.
Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

By using this instruction with TRN2, a $2 \times 2$ matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where $Q = 0$.

![Example Operation of TRN1 and TRN2](image)

Depending on the settings in the $CPACR_{EL1}$, $CPTR_{EL2}$, and $CPTR_{EL3}$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 1  | 0  | size| 0  | 0  | 0  | 1  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |
| op |

Advanced SIMD

TRN1 $<Vd>.<T>, <Vn>.<T>, <Vm>.<T>$

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Vm>$ Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
TRN2

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

By using this instruction with TRN1, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.

![Diagram of TRN1 and TRN2 halfword operations]

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

\[
\begin{align*}
\text{TRN2 } \langle \text{Vd}, \langle T \rangle, \langle Vn \rangle, \langle T \rangle, \langle Vm \rangle, \langle T \rangle
\end{align*}
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{if } \text{size}:Q == '110' \text{ then UNDEFINED; } \\
\text{integer } \text{esize} &= 8 \ll \text{UInt}(\text{size}); \\
\text{integer } \text{datasize} &= \text{if } Q == '1' \text{ then 128 else 64; } \\
\text{integer } \text{elements} &= \text{datasize} \div \text{esize; } \\
\text{integer } \text{part} &= \text{UInt}(\text{op}); \\
\text{integer } \text{pairs} &= \text{elements} \div 2;
\end{align*}
\]

Assembler Symbols

\[
\begin{align*}
\text{<Vd>} & \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.} \\
\text{<T>} & \quad \text{Is an arrangement specifier, encoded in “size:Q”:} \\
\text{<Vn>} & \quad \text{Is the name of the first SIMD&FP source register, encoded in the “Rn” field.} \\
\text{<Vm>} & \quad \text{Is the name of the second SIMD&FP source register, encoded in the “Rm” field.}
\end{align*}
\]

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If the Self-Hosted Trace Extension is not implemented, this instruction executes as a NOP.

System (ARMv8.4)

```
<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

System HintOp op;

```java
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 111' op = SystemHintOp_ESB;
    when '0001 xxx' op = SystemHintOp_PSB;
    when '0010 000'
        if !HaveRASExt() then EndOfInstruction();  // Instruction executes as NOP
        else op = SystemHintOp_ESR;
    when '0010 001'
        if !HaveStatisticalProfiling() then EndOfInstruction();  // Instruction executes as NOP
        else op = SystemHintOp_PSE;
    when '0010 010'
        if !HaveSelfHostedTrace() then EndOfInstruction();  // Instruction executes as NOP
        else op = SystemHintOp_TSB;
    when '0010 100'
        if !HaveSelfHostedTrace() then EndOfInstruction();  // Instruction executes as NOP
        else op = SystemHintOp_CSDB;
    when '0011 xxx'
        if !HaveSelfHostedTrace() then EndOfInstruction();  // Instruction executes as NOP
        else op = SystemHintOp_BTI;
        BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
    when '0100 xxx'
        if !HaveSelfHostedTrace() then EndOfInstruction();  // Instruction executes as NOP
        else op = SystemHintOp_CSDB;
    otherwise EndOfInstruction();  // Instruction executes as NOP
```
case op of
when SystemHintOp_YIELD
    Hint_Yield();
when SystemHintOp_WFE
    if IsEventRegisterSet() then
        ClearEventRegister();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch64.CheckForWFETrap(EL1, TRUE);
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFETrap(EL2, TRUE);
        if HaveEL(EL3) && PSTATE.EL != EL0 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFETrap(EL3, TRUE);
            WaitForEvent();
when SystemHintOp_WFI
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch64.CheckForWFETrap(EL1, FALSE);
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFETrap(EL2, FALSE);
        if HaveEL(EL3) && PSTATE.EL != EL0 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFETrap(EL3, FALSE);
            WaitForInterrupt();
when SystemHintOp_SEV
    SendEvent();
when SystemHintOp_SEVL
    SendEventLocal();
when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
when SystemHintOp_TSB
    TraceSynchronizationBarrier();
when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
when SystemHintOp_BTI
    BTypeNext = '00';
otherwise // do nothing
TST (immediate)

Test bits (immediate), setting the condition flags and discarding the result: \( Rn \ \text{AND} \ \text{imm} \).

This is an alias of \text{ANDS (immediate)}. This means:

- The encodings in this description are named to match the encodings of \text{ANDS (immediate)}.
- The description of \text{ANDS (immediate)} gives the operational pseudocode for this instruction.

\[
\begin{array}{ccccccccccccccccccccccccccc}
\hline
\text{sf} & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & \text{N} & \text{immr} & \text{imms} & \text{Rn} & 1 & 1 & 1 & 1 & 1 & \text{opc} & \text{Rd}
\end{array}
\]

32-bit (\( \text{sf} = 0 \ \&\& \ N = 0 \))

\[
\text{TST} \ <Wn> \ , \ #<\text{imm}>
\]

is equivalent to

\[
\text{ANDS} \ \text{WZR}, \ <Wn>, \ #<\text{imm}>
\]

and is always the preferred disassembly.

64-bit (\( \text{sf} = 1 \))

\[
\text{TST} \ <Xn> \ , \ #<\text{imm}>
\]

is equivalent to

\[
\text{ANDS} \ \text{XZR}, \ <Xn>, \ #<\text{imm}>
\]

and is always the preferred disassembly.

**Assembler Symbols**

- \(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<\text{imm}>\) For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Operation**

The description of \text{ANDS (immediate)} gives the operational pseudocode for this instruction.
**TST (shifted register)**

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of **ANDS (shifted register)**. This means:

- The encodings in this description are named to match the encodings of **ANDS (shifted register)**.
- The description of **ANDS (shifted register)** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>1 1 0 1 0 1 0</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0)

TST \(<Wn>, <Wm>{, <shift> #<amount>}\)

is equivalent to

ANDS WZR, \(<Wn>, <Wm>{, <shift> #<amount>}\)

and is always the preferred disassembly.

### 64-bit (sf == 1)

TST \(<Xn>, <Xm>{, <shift> #<amount>}\)

is equivalent to

ANDS XZR, \(<Xn>, <Xm>{, <shift> #<amount>}\)

and is always the preferred disassembly.

**Assembler Symbols**

- \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<shift>\) Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- \(<amount>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

**Operation**

The description of **ANDS (shifted register)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPUACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembly Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```cpp
checkFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register, while the UABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>0 Q 1 0 1 1 0</th>
<th>size 1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

U

op

Three registers, not all the same type

UABAL(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>

Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>

Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm>

Is the name of the second SIMD&FP source register, encoded in the “Rm” field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three registers of the same type

$UABD <Vd>.<T>$, $<Vn>.<T>$, $<Vm>.<T>$

integer d = $UInt$(Rd);
integer n = $UInt$(Rn);
integer m = $UInt$(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << $UInt$(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');

Assembler Symbols

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Vm>$ Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = $V[n]$;
bits(datasize) operand2 = $V[m]$;
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
result = if accumulate then $V[d]$ else Zeros();
for e = 0 to elements-1
    element1 = $Int$(Elem[operand1, e, esize], unsigned);
    element2 = $Int$(Elem[operand2, e, esize], unsigned);
    absdiff = $Abs$(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
$V[d]$ = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UABDL, UABDL2**

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABDL instruction extracts each source vector from the lower half of each source register, while the UABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | size| 1  | Rm | 0  | 1  | 1  | 0  | Rd | U  | op |

**Three registers, not all the same type**

UABDL(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size == '11' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = 64;
- integer part = UInt(Q);
- integer elements = datasize DIV esize;
- boolean accumulate = (op == '0');
- boolean unsigned = (U == '1');

**Assembler Symbols**

2

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
   element1 = Int(Elem[operand1, e, esize], unsigned);
   element2 = Int(Elem[operand2, e, esize], unsigned);
   absdiff = Abs(element1 - element2)<2*esize-1:0>;
   Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Ta>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- **<Tb>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;
result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UADDL instruction extracts each source vector from the lower half of each source register, while the UADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 1  | 0  | 1  | 1  | 0  | size| 1  | Rm| 0 | 0 | 0 | 0 | 0 | Rn | 0 | 0 | Rd |
|    | 01 |

Three registers, not all the same type

UADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

2    Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>    Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>    Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>    Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb>    Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm>    Is the name of the second SIMD&FP source register, encoded in the “Rm” field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “size:Q”:
    | size | Q | <Ta> |
    |------|---|------|
    | 00   | 0 | 4H   |
    | 00   | 1 | 8H   |
    | 01   | 0 | 2S   |
    | 01   | 1 | 4S   |
    | 10   | 0 | 1D   |
    | 10   | 1 | 2D   |
    | 11   | x | RESERVED |

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “size:Q”:
    | size | Q | <Tb> |
    |------|---|------|
    | 00   | 0 | 8B   |
    | 00   | 1 | 16B  |
    | 01   | 0 | 4H   |
    | 01   | 1 | 8H   |
    | 10   | 0 | 2S   |
    | 10   | 1 | 4S   |
    | 11   | x | RESERVED |
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0</td>
<td>size 1 1 0 0 0 0 0 0 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td>U</td>
</tr>
</tbody>
</table>

Advanced SIMD

UADDLV <V><d>, <Vn><T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);
V[d] = sum<2*esize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The UADDW instruction extracts vector elements from the lower half of the second source register, while the UADDW2 instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | Q  | 1  | 0  | 1  | 1  | 1  | 0  | size | 1  |    |    | Rm | 0  | 0  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Three registers, not all the same type

UADDW(2) <Vd>,<Ta>, <Vn>,<Ta>, <Vm>,<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integerdatasize = 64;
integerpart = UInt(Q);
integerelements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UBFIZ

Unsigned Bitfield Insert in Zeros copies a bitfield of \(<width>\) bits from the least significant bits of the source register to bit position \(<lsb>\) of the destination register, setting the destination bits above and below the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | N  | immr| immr| Rn  | Rd  |

```
opc
```

32-bit (sf == 0 && N == 0)

```
UBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit (sf == 1 && N == 1)

```
UBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler Symbols

- \(<Wd>\): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\): Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\): Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<lsb>\): For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- \(<width>\): For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly. If \(<\text{imms}>\) is greater than or equal to \(<\text{immr}>\), this copies a bitfield of \(<\text{imms}>-<\text{immr}>+1\) bits starting from bit position \(<\text{immr}>\) in the source register to the least significant bits of the destination register.

If \(<\text{imms}>\) is less than \(<\text{immr}>\), this copies a bitfield of \(<\text{imms}>+1\) bits from the least significant bits of the source register to bit position \((\text{regsize}-<\text{immr}>)\) of the destination register, where \text{regsize} is the destination register size of 32 or 64 bits.

In both cases the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases \texttt{LSL (immediate)}, \texttt{LSR (immediate)}, \texttt{UBFIZ}, \texttt{UBFX}, \texttt{UXTB}, and \texttt{UXTH}.

### Assembler Symbols

- \texttt{<Wd>} is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \texttt{<Wn>} is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \texttt{<Xd>} is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \texttt{<Xn>} is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \texttt{<immr>} for the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.
- \texttt{<immr>} for the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
- \texttt{<imms>} for the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.
- \texttt{<imms>} for the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 1 0 1 0 0 1 1 N immr imms Rn Rd

opc

32-bit (sf == 0 && N == 0)

UBFM <Wd>, <Wn>, #<immr>, #<imms>

64-bit (sf == 1 && N == 1)

UBFM <Xd>, <Xn>, #<immr>, #<imms>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

case opc of
  when '00' inzero = TRUE; extend = TRUE; // SBFM
  when '01' inzero = FALSE; extend = FALSE; // BFM
  when '10' inzero = TRUE; extend = FALSE; // UBFM
  when '11' UNDEFINED;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```
### Alias Conditions

<table>
<thead>
<tr>
<th>Alias (immediate)</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL (immediate)</td>
<td>32-bit</td>
<td>imms != '011111' &amp;&amp; imms + 1 == immr</td>
</tr>
<tr>
<td>LSL (immediate)</td>
<td>64-bit</td>
<td>imms != '111111' &amp;&amp; imms + 1 == immr</td>
</tr>
<tr>
<td>LSR (immediate)</td>
<td>32-bit</td>
<td>imms == '011111'</td>
</tr>
<tr>
<td>LSR (immediate)</td>
<td>64-bit</td>
<td>imms == '111111'</td>
</tr>
<tr>
<td>UBFIZ</td>
<td></td>
<td>UInt(imms) &lt; UInt(immr)</td>
</tr>
<tr>
<td>UBFX</td>
<td></td>
<td>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</td>
</tr>
<tr>
<td>UXTB</td>
<td></td>
<td>immr == '000000' &amp;&amp; imms == '000111'</td>
</tr>
<tr>
<td>UXTH</td>
<td></td>
<td>immr == '000000' &amp;&amp; imms == '001111'</td>
</tr>
</tbody>
</table>

### Operation

```
bits(dataSize) dst = if inzero then Zeros() else X[d];
bits(dataSize) src = X[n];

// perform bitfield move on low bits
bits(dataSize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(dataSize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UBFX**

Unsigned Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of **UBFM**. This means:

- The encodings in this description are named to match the encodings of **UBFM**.
- The description of **UBFM** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0 && N == 0)

UBFX <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

UBFM <Wd>, <Wn>, #<lsb>, #{<lsb>+<width>-1}

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

### 64-bit (sf == 1 && N == 1)

UBFX <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

UBFM <Xd>, <Xn>, #<lsb>, #{<lsb>+<width>-1}

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<lsb>** For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
  - For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- **<width>** For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  - For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

**Operation**

The description of **UBFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UCVTF (vector, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 1 1 1 1 1 1 1 0 != 0000 immh 1 1 0 0 1 Rn Rd</td>
</tr>
<tr>
<td>Uimmh</td>
</tr>
</tbody>
</table>

Scalar

UCVTF <V><d>, <V><n>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 1 0 != 0000 immh 1 1 0 0 1 Rn Rd</td>
</tr>
<tr>
<td>Uimmh</td>
</tr>
</tbody>
</table>

Vector

UCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdim);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits>
For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAvS1MDEnabled64();
bits(datasize) operand = V[n];
bios(datasize) result;
bios(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UCVTF (vector, integer)**

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UCVTF <Hd>, <Hn>**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UCVTF <V>d>, <V>n>**

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

### Vector half precision

(ARMv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector half precision

UCVTF <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);
V[d] = result;
```
UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

```
   31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
   sf  0  0  1  1  1  1  0  type  0  0  0  0  1  1  scale  Rn  Rd
   rmode  opcode
```
32-bit to half-precision (sf == 0 && type == 11) (ARMv8.2)

```
UCVTF <Hd>, <Wn>, #<fbits>
```

32-bit to single-precision (sf == 0 && type == 00)

```
UCVTF <Sd>, <Wn>, #<fbits>
```

32-bit to double-precision (sf == 0 && type == 01)

```
UCVTF <Dd>, <Wn>, #<fbits>
```

64-bit to half-precision (sf == 1 && type == 11) (ARMv8.2)

```
UCVTF <Hd>, <Xn>, #<fbits>
```

64-bit to single-precision (sf == 1 && type == 00)

```
UCVTF <Sd>, <Xn>, #<fbits>
```

64-bit to double-precision (sf == 1 && type == 01)

```
UCVTF <Dd>, <Xn>, #<fbits>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  end

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
  when '00 11'        // FCVTZ
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00'        // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  otherwise
    UNDEFINED;
end

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits> For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".
For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
    V[d] = fltval;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the \textit{FPCR}, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR EL1}, \textit{CPTR EL2}, and \textit{CPTR EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
sf 0 0 1 1 1 0 type 1 0 0 0 1 1 0 0 0 0 0 0 Rd
\end{verbatim}

\textit{sf}: Status flag
\textit{type}: Instruction type
\textit{Rn}: General-purpose source register
\textit{Rd}: SIMD&FP destination register
32-bit to half-precision (sf == 0 && type == 11) (ARMv8.2)

UCVT <Hd>, <Wn>

32-bit to single-precision (sf == 0 && type == 00)

UCVT <Sd>, <Wn>

32-bit to double-precision (sf == 0 && type == 01)

UCVT <Dd>, <Wn>

64-bit to half-precision (sf == 1 && type == 11) (ARMv8.2)

UCVT <Hd>, <Xn>

64-bit to single-precision (sf == 1 && type == 00)

UCVT <Sd>, <Xn>

64-bit to double-precision (sf == 1 && type == 01)

UCVT <Dd>, <Xn>
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UNDEFINED;
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    case opcode<2:1>:rmode of
        when '00 xx'        // FCVT[NPMZ][US]
            rounding = FPDecodeRounding(rmode);
            unsigned = (opcode<0> == '1');
            op = FPConvOp_CVT_FtoI;
        when '01 00'        // [US]CVTF
            rounding = FPRoundingMode(FPCR);
            unsigned = (opcode<0> == '1');
            op = FPConvOp_CVT_ItoF;
        when '10 00'        // FCVTA[US]
            rounding = FPRounding_TIEAWAY;
            unsigned = (opcode<0> == '1');
            op = FPConvOp_CVT_FtoI;
        when '11 00'        // FMOV
            if fltsize != 16 && fltsize != intsize then UNDEFINED;
            op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
            part = 0;
        when '11 01'        // FMOV D[1]
            if intsize != 64 || fltsize != 128 then UNDEFINED;
            op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
            part = 1;
            fltsize = 64; // size of D[1] is 64
        when '11 11'        // FJCVTZS
            if !HaveFJCVTZSExt() then UNDEFINED;
            rounding = FPRounding_ZERO;
            unsigned = (opcode<0> == '1');
            op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
```
UDF

Permanently Undefined generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for UDF used in this section are defined as permanently UNDEFINED in the ARMv8-A architecture.

Integer

UDF #<imm>

// The imm16 field is ignored by hardware.
UNDEFINED;

Assembler Symbols

<imm> is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

Operation

// No operation.
UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rd</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

UDIV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

UDIV <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean unsigned = (o1 == '0');

Assembler Symbols

<Wr>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>  Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>  Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xr>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>  Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
  result = 0;
else
  result = RoundTowardsZero(Real(Int(operand1, unsigned)) / Real(Int(operand2, unsigned)));

X[d] = result<datasize-1:0>;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UDOT (by element)

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an \texttt{OPTIONAL} instruction. From ARMv8.4 it is mandatory for all implementations to support it. \texttt{ID\_AA64ISAR0\_EL1.DP} indicates whether this instruction is supported.

\textbf{Vector (ARMv8.2)}

\begin{center}
\begin{tabular}{cccccccccccccccc}
0 & Q & 1 & 0 & 1 & 1 & 1 & size & L & M & Rm & 1 & 1 & 1 & 0 & H & 0 & Rn & Rd \\
\hline
\end{tabular}
\end{center}

\textbf{Vector}

\[ \text{UDOT} <\text{Vd}>.Q<Ta>, <\text{Vn}>.Q<Tb>, <\text{Vm}>.4B[<\text{index}>] \]

\begin{verbatim}
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U=='0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
\end{verbatim}

\begin{flushleft}
\textbf{Assembler Symbols}
\end{flushleft}

\textbf{<Vd>}  
Is the name of the SIMD\&FP destination register, encoded in the "Rd" field.

\textbf{<Ta>}  
Is an arrangement specifier, encoded in “Q”:

\begin{center}
\begin{tabular}{c|c}
Q & <Ta> \\
\hline
0 & 2S \\
1 & 4S \\
\end{tabular}
\end{center}

\textbf{<Vn>}  
Is the name of the first SIMD\&FP source register, encoded in the "Rn" field.

\textbf{<Tb>}  
Is an arrangement specifier, encoded in “Q”:

\begin{center}
\begin{tabular}{c|c}
Q & <Tb> \\
\hline
0 & 8B \\
1 & 16B \\
\end{tabular}
\end{center}

\textbf{<Vm>}  
Is the name of the second SIMD\&FP source register, encoded in the "M:Rm" fields.

\textbf{<index>}  
Is the element index, encoded in the "H:L" fields.
Operation

```c
CheckFPAdvSIMDEnabled64();
bots(datasize) operand1 = V[n];
bots(128) operand2 = V[m];
bots(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        end
        element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
        res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UDOT (vector)

Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it. ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Three registers of the same type
(ARMv8.2)

```
<table>
<thead>
<tr>
<th>Q</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Three registers of the same type

UDOT `<Vd>..<Ta>`, `<Vn>..<Tb>`, `<Vm>..<Tb>`

```
if !HaveDOTPExt() then UNDEFINED;
if size! = '10' then UNDEFINED;
boolean signed = (U=='0');
integer d = UINT(Rd);
integer n = UINT(Rn);
integer m = UINT(Rm);
integer esize = 8 << UINT(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ta>` Is an arrangement specifier, encoded in “Q”:
  - `Q` `<Ta>`
    - 0: 2S
    - 1: 4S
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>` Is an arrangement specifier, encoded in “Q”:
  - `Q` `<Tb>`
    - 0: 8B
    - 1: 16B
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```
**UHADD**

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see **URHADD**.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Three registers of the same type

UHADD `<Vd>..<T>`, `<Vn>..<T>`, `<Vm>..<T>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in “size-Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  sum = element1 + element2;
  Elem[result, e, esize] = sum<esize:1>;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
UHSUB

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three registers of the same type

UHSUB `<Vd>`.<`<T>`>, `<Vn>`.<`<T>`>, `<Vm>`.<`<T>`>

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<<T>` Is an arrangement specifier, encoded in “size-Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th><code>&lt;&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
   element1 = Int(Elem[operand1, e, esize], unsigned);
   element2 = Int(Elem[operand2, e, esize], unsigned);
   diff = element1 - element2;
   Elem[result, e, esize] = diff<esize:1>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias **UMULL**.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>Ra</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit

**UMADDL** <Xd>, <Wn>, <Wm>, <Xa>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m =UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

- **<Xd>** is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- **<Xa>** is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UMULL</strong></td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

if sub_op then
    result = Int(operand3, unsigned) - (Int(operand1, unsigned) * Int(operand2, unsigned));
else
    result =Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

**Operational information**

If **PSTATE.DIT** is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMAX

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>Rm</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

```
integer integer d = UInt(Rd);
integer integer n = UInt(Rn);
integer integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer integer esize = 8 << UInt(size);
integer integer datasize = if Q == '1' then 128 else 64;
integer integer elements = datasize DIV esize;

boolean boolean unsigned = (U == '1');
boolean boolean minimum = (o1 == '1');
```

```c
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMAXP**

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Three registers of the same type**

UMAXP `<Vd>`.<`<T>`>, `<Vn>`.<`<T>`>, `<Vm>`.<`<T>`>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<<T>>` Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th><code>&lt;&lt;T&gt;&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
  element1 = Int(Elem[concat, 2*e, esize], unsigned);
  element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UMAXV**

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| U | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | Rd |

Advanced SIMD

**UMAXV <V><d>, <Vn>.<T>**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean min = (op == '1');
```

**Assembler Symbols**

**<V>** Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<d>** Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

**<Vn>** Is the name of the SIMD&FP source register, encoded in the “Rn” field.

**<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;
maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d] = maxmin<esize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Three registers of the same type

<table>
<thead>
<tr>
<th>UMIN</th>
<th>(&lt;Vd&gt;), (&lt;T&gt;), (&lt;Vn&gt;), (&lt;T&gt;), (&lt;Vm&gt;), (&lt;T&gt;)</th>
</tr>
</thead>
</table>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

### Assembler Symbols

- \(<Vd>\)** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- \(<T>\)** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- \(<Vn>\)** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- \(<Vm>\)** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```plaintext
checkFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Assembler Symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in “size:Q”:
  ```
  size  Q  <T>
  00  0  8B
  00  1  16B
  01  0  4H
  01  1  8H
  10  0  2S
  10  1  4S
  11  x  RESERVED
  ```
  - `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
  - `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMINV

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

`UMINV <V><d>, <Vn>.<T>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:
```plaintext
size  <V>
00    B
01    H
10    S
11    RESERVED
```

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:
```plaintext
size  Q  <T>
00    0    8B
00    1    16B
01    0    4H
01    1    8H
10    0    RESERVED
10    1    4S
11    x    RESERVED
```

Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;
maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
   element = Int(Elem[operand, e, esize], unsigned);
   maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d] = maxmin<esize-1:0>;
```

UMINV
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMLAL, UMLAL2 (by element)

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register, while the UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
vector

UMLAL[2] <Vd>, <Ta>, <Vn>, <Tb>, <Vm>, <Ts>[<index>]
```

integer idxxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
  case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
  integer d = UInt(Rd);
  integer n = UInt(Rn);
  integer m = UInt(Rmhi:Rm);
  integer esize = 8 << UInt(size);
  integer datasize = 64;
  integer part = UInt(Q);
  integer elements = datasize DIV esize;
  boolean unsigned = (U == '1');
  boolean sub_op = (o2 == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:
### Size Quarters

<table>
<thead>
<tr>
<th>Size</th>
<th>Q</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### Vm

Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>Size</th>
<th>Vm</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

### Ts

Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>Size</th>
<th>Ts</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### Index

Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>Size</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elm[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elm[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elm[result, e, 2*esize] = Elm[operand3, e, 2*esize] - product;
    else
        Elm[result, e, 2*esize] = Elm[operand3, e, 2*esize] + product;
V[d] = result;
```

### Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register, while the UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type

```
UMLAL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Assembler Symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in “size”:  

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLSL instruction extracts vector elements from the lower half of the first source register, while the UMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th></th>
<th></th>
<th>H</th>
<th></th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector

UMLSL[2] <Vd>, <Ta>, <Vn>, <Tb>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');

Assembler Symbols

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th></th>
<th></th>
<th>H</th>
<th></th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
<td></td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMLSL instruction extracts each source vector from the lower half of each source register, while the UMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Three registers, not all the same type

UMLSL(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMNEGL**

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of **UMSUBL**. This means:

- The encodings in this description are named to match the encodings of **UMSUBL**.
- The description of **UMSUBL** gives the operational pseudocode for this instruction.

![Register Codes](image)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 64-bit

**Assembler Symbols**

- **<Xd>** is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

The description of **UMSUBL** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (to general).

32-bit (Q == 0)

UMOV <Wd>, <Vn>.<Ts>[<index>]

64-reg,UMOV-64-reg (Q == 1 && imm5 == x1000)

UMOV <Xd>, <Vn>.<Ts>[<index>]

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
  when '0xxxx1' size = 0; // UMOV Wd, Vn.B
  when '0xxx10' size = 1; // UMOV Wd, Vn.H
  when '0xx100' size = 2; // UMOV Wd, Vn.S
  when '1x1000' size = 3; // UMOV Xd, Vn.D
  otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
</tbody>
</table>

For the 64-reg,UMOV-64-reg variant: is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x00000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xx100</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the 32-bit variant: is the element index encoded in “imm5”:
### Aliases Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (to general)</td>
<td>imm5 == ‘x1000‘</td>
</tr>
<tr>
<td>MOV (to general)</td>
<td>imm5 == ‘xx100‘</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMEnabled64();
blts(idxdsiz) operand = V[n];
X[d] = ZeroExtend(Elem[operand, index, esize], datasize);
```

### Operational Information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

---

For the 64-reg, UMOV-64-reg variant: is the element index encoded in "imm5<4>".

---

Reserved
Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias **UMNEGL**.

### 64-bit

**UMSUBL**<br>
\(<X_d>, <W_n>, <W_m>, <X_a>\)

\[
\begin{align*}
\text{integer } d &= \text{UInt}(R_d); \\
\text{integer } n &= \text{UInt}(R_n); \\
\text{integer } m &= \text{UInt}(R_m); \\
\text{integer } destsize &= 64; \\
\text{integer } datasetize &= 32; \\
\text{boolean } \text{sub_op} &= (o0 == '1'); \\
\text{boolean } \text{unsigned} &= (U == '1');
\end{align*}
\]

**Assembler Symbols**

- **<X_d>** is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<W_n>** is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<W_m>** is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- **<X_a>** is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UMNEGL</strong></td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

\[
\begin{align*}
\text{bits}(datasetize) \phantom{|} \text{operand1} &= X[n]; \\
\text{bits}(datasetize) \phantom{|} \text{operand2} &= X[m]; \\
\text{bits}(destsize) \phantom{|} \text{operand3} &= X[a];
\end{align*}
\]

\[
\begin{align*}
\text{integer } \text{result}; \\
\text{if } \text{sub_op} \text{ then} \\
\quad \text{result} &= \text{Int}(\text{operand3, unsigned}) - (\text{Int}(\text{operand1, unsigned}) \times \text{Int}(\text{operand2, unsigned})); \\
\text{else} \\
\quad \text{result} &= \text{Int}(\text{operand3, unsigned}) + (\text{Int}(\text{operand1, unsigned}) \times \text{Int}(\text{operand2, unsigned})); \\
X[d] &= \text{result}_{63:0};
\end{align*}
\]

**Operational information**

If **PSTATE.DIT** is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
 UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | 0  | (1) | (1) | (1) | (1) |
| U   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Ra  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

64-bit

UMULH <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);  // ignored by UMULH/SMULH
integer destsize = 64;
datasize = destsize;
boolean unsigned = (U == '1');

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

integer result;

result = Int(operand1, unsigned) * Int(operand2, unsigned);

X[d] = result<127:64>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMULL, UMULL2 (by element)**

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The **UMULL** instruction extracts vector elements from the lower half of the first source register, while the **UMULL2** instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | size | L  | M  | Rm | 1  | 0  | 1  | 0  | H  | 0  | Rn | Rd |

**Vector**

\[
\text{UMULL[2]} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
\]

integer idxxdszie = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = \text{UInt}(H:L:M); Rmhi = '0';
  when '10' index = \text{UInt}(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rmhi:Rm);
integer esize = 8 << \text{UInt}(size);
integer datasize = 64;
integer part = \text{UInt}(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

**Assembler Symbols**

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{absent}</td>
</tr>
<tr>
<td>1</td>
<td>{present}</td>
</tr>
</tbody>
</table>

\(<Vd>\)

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Ta>\)

Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;Ta&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vn>\)

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\)

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>(&lt;Tb&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vm>\)

Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:
Restricted to V0-V15 when element size \(<Ts>\) is H.

<index>  
Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;\text{index}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = \text{Int(Elem[operand2, index, esize], unsigned});
for e = 0 to \text{elements-1}
    element1 = \text{Int(Elem[operand1, e, esize], unsigned});
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMULL instruction extracts each source vector from the lower half of each source register, while the UMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|0  |Q |1 |0 |1 |1 |1 |0 |size|1 |Rm |1 |1 |0 |0 |0 |Rn |Rd |

Three registers, not all the same type

UMULL(2)  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);  
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2. Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt; Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  Elem[result, e, 2*esize] = (element1 * element2)<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of UMADDL. This means:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | Rm | 0 | 1 | 1 | 1 | 1 | Rd |

64-bit

UMULL <Xd>, <Wn>, <Wm>

is equivalent to

UMADDL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UQADD

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \( FPSR.QC \) is set.

Depending on the settings in the \( CPACR_EL1, CPTR_EL2, \) and \( CPTR_EL3 \) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 1 1 1 0 | size | 1 | Rm | 0 0 0 0 1 1 | Rn | Rd |

\( U \)

**Scalar**

\[ UQADD \langle V\rangle<d>, \langle V\rangle<n>, \langle V\rangle<m> \]

```plaintext
integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer esize = 8 << Uint(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

**Vector**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 Q 1 0 1 1 1 0 | size | 1 | Rm | 0 0 0 0 1 1 | Rn | Rd |

\( U \)

**Vector**

\[ UQADD \langle Vd\rangle.<T>, \langle Vn\rangle.<T>, \langle Vm\rangle.<T> \]

```plaintext
integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << Uint(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

**Assembler Symbols**

\(<V>\)

Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\)

Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\)

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<m>\)

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\(<Vd>\)

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the ”Rm” field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see _UQSHL_.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit _FPSR_.QC is set.

Depending on the settings in the _CPACR_EL1_, _CPTR_EL2_, and _CPTR_EL3_ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

```
0 1 1 1 1 1 0  size 1  Rm 0 1 0 1 1 1  Rn  Rd
0 1 1 1 1 0  size 1  Rm 0 1 0 1 1 1  Rn  Rd
U  R  S
```

**UQRSHL <V><d>, <V><n>, <V><m>**

- integer \(d = \text{UInt}(Rd)\);
- integer \(n = \text{UInt}(Rn)\);
- integer \(m = \text{UInt}(Rm)\);
- integer esize = 8 \(<\text{UInt}(size)\);
- integer datasize = esize;
- integer elements = 1;
- boolean unsigned = (U == '1');
- boolean rounding = (R == '1');
- boolean saturating = (S == '1');
- if S == '0' && size != '11' then UNDEFINED;

### Vector

```
0  Q 1 0 1 1 1 0  size 1  Rm 0 1 0 1 1 1  Rn  Rd
0  Q 1 0 1 1 1 0  size 1  Rm 0 1 0 1 1 1  Rn  Rd
U  R  S
```

**UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

- integer \(d = \text{UInt}(Rd)\);
- integer \(n = \text{UInt}(Rn)\);
- integer \(m = \text{UInt}(Rm)\);
- if size:Q == '110' then UNDEFINED;
- integer esize = 8 \(<\text{UInt}(size)\);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;
- boolean unsigned = (U == '1');
- boolean rounding = (R == '1');
- boolean saturating = (S == '1');

**Assembler Symbols**

- `<V>` is a width specifier, encoded in “size”:
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m>
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    else
        element = Elem[operand1, e, esize];
    if saturating then
        Elem[result, e, esize] = SatQ(element, esize, unsigned);
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQRSHRN, UQRSHRN2

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see UQSHRN.

The UQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

UQRSHRN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

Vector

UQRSHRN[2] <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding
the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>lxxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>lxxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb>

Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>lxxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>

Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va>

Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>lxxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift>

For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>16-UInt(immh:immb)</td>
</tr>
<tr>
<td>001x</td>
<td>32-UInt(immh:immb)</td>
</tr>
<tr>
<td>01xx</td>
<td>64-UInt(immh:immb)</td>
</tr>
<tr>
<td>lxxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>16-UInt(immh:immb)</td>
</tr>
<tr>
<td>001x</td>
<td>32-UInt(immh:immb)</td>
</tr>
<tr>
<td>01xx</td>
<td>64-UInt(immh:immb)</td>
</tr>
<tr>
<td>lxxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
  element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
  (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
  if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```
UQSHL (immediate)

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 0  | 1  | 1  | 0  | 1  | Rd  |
| U  | immh | op |

Scalar

UQSHL <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
endcase;

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | != 0000 | immh | 0  | 1  | 1  | 0  | 1  | Rd  |
| U  | immh | op |
Vector

UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE;  dst_unsigned = TRUE;

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMEnabled64();

bits(datasize) operand = V[n];

bits(datasize) result;

integer element;

boolean sat;

for e = 0 to elements-1
  element = Int(Elem[operand, e, esize], src_unsigned) << shift;
  (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
  if sat then FPSR.QC = '1';

  V[d] = result;
UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0   1 1 1 1 1 1 0 | size | 1               | Rm              | 0   1 0 0 1 1 1 | Rn              | Rd              |
| U   | R   | S               |                 |                 |                 |

Scalar

UQSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size == '11' then UNDEFINED;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0   Q 1 0 1 1 1 1 0 | size | 1               | Rm              | 0   1 0 0 1 1 1 | Rn              | Rd              |
| U   | R   | S               |                 |                 |                 |

Vector

UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”: 

UQSHL (register)
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m>
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>Size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>01</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    else
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSHRN, UQSHRN2

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see UQRSHRN.

The UQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | !=0000| immh | 1  | 0  | 0  | 1  | 0  | 1  | Rn | Rd |
| U  | immh| op |

Scalar

UQSHRN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | !=0000| immh | 1  | 0  | 0  | 1  | 0  | 1  | Rn | Rd |
| U  | immh| op |

Vector

UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

\(<Vd>\)
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Tb>\)
Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>(&lt;Tb&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vn>\)
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<Ta>\)
Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;Ta&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vb>\)
Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;Vb&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<d>\)
Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<Va>\)
Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;Va&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<n>\)
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<shift>\)
For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;shift&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>(&lt;shift&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-\text{UInt}(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```plaintext
CheckFPAdvSIMEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
(Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSUB

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |
| U  |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |

Scalar

UQSUB <V><d>, <V><n>, <V><m>

| integer d = UInt(Rd) |
| integer n = UInt(Rn) |
| integer m = UInt(Rm) |
| integer esize = 8 << UInt(size) |
| integer datasize = esize |
| integer elements = 1 |
| boolean unsigned = (U == '1') |

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |
| U  |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |

Vector

UQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

| integer d = UInt(Rd) |
| integer n = UInt(Rn) |
| integer m = UInt(Rm) |
| if size:Q == '110' then UNDEFINED |
| integer esize = 8 << UInt(size) |
| integer datasize = if Q == '1' then 128 else 64 |
| integer elements = datasize DIV esize |
| boolean unsigned = (U == '1') |

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

<Vn> is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The **UQXTN** instruction writes the vector to the lower half of the destination register and clears the upper half, while the **UQXTN2** instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |
```

**UQXTN** <\(V_b\)<d>, <\(V_a\)<n>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |
```

**UQXTN**(2) <\(V_d\).<\(T_b\>, <\(V_n\).<\(T_a\>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:
Q 2
0 [absent]
1 [present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URECPE**

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  |

**Vector**

**URECPE** `<Vd>.<T>`, `<Vn>.<T>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

**Assembler Symbols**

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;
for e = 0 to elements-1
  element = Elem[operand, e, 32];
  Elem[result, e, 32] = UnsignedRecipEstimate(element);
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URHADD**

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see **UHADD**.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Binary representation of the instruction](#)

Three registers of the same type

**URHADD** `<Vd>..<T>`, `<Vn>..<T>`, `<Vm>..<T>`

integer d  = UInt(Rd);
integer n  = UInt(Rn);
integer m  = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

**Assembler Symbols**

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

`<Vn>`  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>`  
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  Elem[result, e, esize] = (element1 + element2 + 1)<<esize:1>;
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00ber8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0   1   1   1   1   1   0   size   1   Rm   | 0   1   0   1   0   1   Rd   |
| U   | R   | S   |

Scalar

URSHL <V><d>, <V><n>, <V><m>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' & size != '11' then UNDEFINED;
```

### Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0   1   0   1   1   1   0   size   1   Rm   | 0   1   0   1   0   1   Rd   |
| U   | R   | S   |

Vector

URSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in &quot;size&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

| <d> | Is the number of the SIMD&FP destination register, in the "Rd" field. |
\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
\(<m>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<V_n>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\(<V_m>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
b(I)ts(datasize) operand1 = \(V[n]\);
b(I)ts(datasize) operand2 = \(V[m]\);
b(I)ts(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
        else
            Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8 rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URSHR**

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see **USHR**.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td></td>
<td>immb</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Scalar**

URSHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

**Vector**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td></td>
<td>immb</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Vector**

URSHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asidimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

**Assembler Symbols**

<V> Is a width specifier, encoded in “immh”:
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift>
For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bids(datasize) operand = V[n];
bids(datasize) operand2;
bids(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UNSQRTE

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0  Q 1 0 1 1 0 1 sz 1 0 0 0 0 1 1 0 0 1 0     Rn     Rd
```

Vector

```
UNSQRTE <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in “sz:Q”:
  - `sz` indicates the size of the elements (1 or 2).
  - `Q` indicates the alignment of the elements (1 or 4).
  - For other values, `RESERVED`.

- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRSqrtEstimate(element);

V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
URSRA

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | != | 0000| immh| 0  | 0  | 1  | 1  | 0  | 1  | Rn | | Rd |
| U  | immh| o1 | o0 |

Scalar

URSRA \(<V>d\), \(<V>n\), \,#\(<shift>\)

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 1  | 0  | != | 0000| immh| 0  | 0  | 1  | 1  | 0  | 1  | Rn | | Rd |
| U  | immh| o1 | o0 |

Vector

URSRA \(<V>d>.<T>\), \(<V>n>.<T>\), \,#\(<shift>\)

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

\(<V>\) Is a width specifier, encoded in “immh”:
Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bias(datasize) operand2; bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USHL

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see USHRL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
<table>
<thead>
<tr>
<th>Rm</th>
<th>Rd</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
U R S
```

Scalar

```
USHL <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

```
<table>
<thead>
<tr>
<th>Q</th>
<th>Rd</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
U R S
```

Vector

```
USHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

V Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
    V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**USHLL, USHLL2**

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The **USHLL** instruction extracts vector elements from the lower half of the source register, while the **USHLL2** instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias **UXTL, UXTL2**.

![Vector Diagram](image)

---

**Assembler Symbols**

| 2 | Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{absent}</td>
</tr>
<tr>
<td>1</td>
<td>{present}</td>
</tr>
</tbody>
</table>

| <Vd> | Is the name of the SIMD&FP destination register, encoded in the "Rd" field. |
|<Ta> | Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

| <Vn> | Is the name of the SIMD&FP source register, encoded in the "Rn" field. |
|<Tb> | Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
</tr>
</tbody>
</table>

| <shift> | Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:

![Vector Diagram](image)
**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UXTL, UXTL2</td>
<td>( \text{immh} == '000' &amp;&amp; \text{BitCount}(\text{immh}) == 1 )</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**USHR**

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see **URSHR**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | ! = 0000 | immb | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| U  | immh | o1 | o0 |

**Scalar**

\[
\begin{align*}
\text{USHR} & \quad \langle V \rangle \langle d \rangle, \quad \langle V \rangle \langle n \rangle, \quad \#\langle \text{shift} \rangle \\
\text{integer } d &= \text{UInt}(Rd) \\
\text{integer } n &= \text{UInt}(Rn) \\
\text{if } \text{immh}[3] \neq '1' \text{ then UNDEFINED;} \\
\text{integer } \text{esize} &= 8 \ll 3 \\
\text{integer } \text{datasize} &= \text{esize} \\
\text{integer } \text{elements} &= 1 \\
\text{integer } \text{shift} &= (\text{esize} \times 2) - \text{UInt}(\text{immh:immh}) \\
\text{boolean unsigned} &= (U == '1') \\
\text{boolean round} &= (o1 == '1') \\
\text{boolean accumulate} &= (o0 == '1')
\end{align*}
\]

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | ! = 0000 | immb | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| U  | immh | o1 | o0 |

**Vector**

\[
\begin{align*}
\text{USHR} & \quad \langle Vd \rangle.\langle T \rangle, \quad \langle Vn \rangle.\langle T \rangle, \quad \#\langle \text{shift} \rangle \\
\text{integer } d &= \text{UInt}(Rd) \\
\text{integer } n &= \text{UInt}(Rn) \\
\text{if } \text{immh} == '0000' \text{ then SEE(asimdimm);} \\
\text{if } \text{immh}<3>:Q == '10' \text{ then UNDEFINED;} \\
\text{integer } \text{esize} &= 8 \ll \text{HighestSetBit}(\text{immh}) \\
\text{integer } \text{datasize} &= \text{if } Q == '1' \text{ then 128 else 64} \\
\text{integer } \text{elements} &= \text{datasize} \div \text{esize} \\
\text{integer } \text{shift} &= (\text{esize} \times 2) - \text{UInt}(\text{immh:immh}) \\
\text{boolean unsigned} &= (U == '1') \\
\text{boolean round} &= (o1 == '1') \\
\text{boolean accumulate} &= (o0 == '1')
\end{align*}
\]

**Assembler Symbols**

\[
\langle V \rangle \quad \text{Is a width specifier, encoded in “immh”}
\]
Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
<td></td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
<td></td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
  element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USQADD

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

Scalar

USQADD <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << Int(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');

Vector

Vector

USQADD <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << Int(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in &quot;size&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;d&gt;</th>
<th>Is the number of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;n&gt;</td>
<td>Is the number of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
</tbody>
</table>
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size-Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see USRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | !=0000| immh | 0  | 0  | 0  | 1  | 0  | 1  | Rn | Rd |

U immh o1 o0

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 1  | 1  | 0  | !=0000| immh | 0  | 0  | 0  | 1  | 0  | 1  | Rn | Rd |

U immh o1 o0

### Assembler Symbols

<V> Is a width specifier, encoded in “immh”:
Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
element = (Int(Elem(operand, e, esize), unsigned) + round_const) >> shift;
Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The USUBL instruction extracts each source vector from the lower half of each source register, while the USUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 1  | 0  | 1  | 1  | 0  | size | 1  | Rm | 0  | 0  | 1  | 0  | 0  | Rn | Rd |
|    |    |    |    |    |    |    |      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Three registers, not all the same type

USUBL(2) <Vd>,<Ta>, <Vn>,<Tb>, <Vm>,<Tb>

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << Uint(size);
integer datasize = 64;
integer part = Uint(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**USUBW, USUBW2**

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. The USUBW instruction extracts vector elements from the lower half of the first source register, while the USUBW2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the **CPACR_EL1, CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | size | 1  | Rm | 0  | 0  | 1  | 1  | 0  | 0  | Rn | 0  | Rd |

**Three registers, not all the same type**

USUBW<2> <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

USUBW, USUBW2 Page 1578
Operation

```cpp
CheckFPAdvSIMEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, 2*esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | Rn |
| sf | opc| N  | immr| immrs|

32-bit

UXTB <Wd>, <Wn>

is equivalent to

UBFM <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of **UBFM**. This means:

- The encodings in this description are named to match the encodings of **UBFM**.
- The description of **UBFM** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf  opc          N  immr       imms</td>
</tr>
</tbody>
</table>

### 32-bit

UXTH `<Wd>, <Wn>`

is equivalent to

UBFM `<Wd>, <Wn>, #0, #15`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

The description of **UBFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UXTL, UXTL2

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. The UXTL instruction extracts vector elements from the lower half of the source register, while the UXTL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of USHLL, USHLL2. This means:

- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

```
+-----------+--------------+----------+
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
+-----------+--------------+----------+
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | ! = 0000 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | Rn | Rd |
+-----------+--------------+----------+
| U | immh | immb |
```

**Vector**


is equivalent to

USHLL[2] <Vd>.<Ta>, <Vn>.<Tb>, #0

and is the preferred disassembly when BitCount(immh) == 1.

**Asmble Styles**

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in &quot;Q&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

```
<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>01x</td>
<td>4S</td>
</tr>
<tr>
<td>0lx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

```
<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>0lx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0lx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

**Operation**

The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UZP1

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can be used with UZP2 to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.

```
Vr: A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0
Vn: B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0
```

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 Q 0 0 1 1 1 0 size 0 Rm 0 0 0 1 1 0 Rn Rd
```

Advanced SIMD

UZP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
  Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UZP2**

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can be used with UZP1 to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.

![Example of UZP1 and UZP2 operation]

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Advanced SIMD

UZP2 \(<\text{Vd}>.\langle T\rangle, \langle T\rangle, \langle T\rangle, \langle T\rangle, \langle Vn\rangle.\langle T\rangle, \langle Vm\rangle.\langle T\rangle\)

- integer \(d = \text{UInt}(Rd)\);
- integer \(n = \text{UInt}(Rn)\);
- integer \(m = \text{UInt}(Rm)\);

if \(\text{size}:Q == '110'\) then UNDEFINED;

- integer \(\text{esize} = 8 \ll \text{UInt}(\text{size})\);
- integer \(\text{datasize} = \text{if } Q == '1' \text{ then 128 else 64} \);
- integer \(\text{elements} = \text{datasize DIV } \text{esize}\);
- integer \(\text{part} = \text{UInt}(\text{op})\);

### Assembler Symbols

- \(<\text{Vd}>\) is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<\langle T\rangle>\) is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>(&lt;\text{T}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- \(<\text{Vn}>\) is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \(<\text{Vm}>\) is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the \texttt{SEV} instruction on any PE in the multiprocessor system. For more information, see \textit{Wait For Event mechanism and Send event}.

As described in \textit{Wait For Event mechanism and Send event}, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- Traps to EL1 of EL0 execution of WFE and WFI instructions.
- Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions.
- Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions.

```c
System

WFE

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111' 
    SEE "XPACLRI";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction();          // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction();    // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
    op = SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction();                              // Instruction executes as NOP
```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFETrap(EL1, TRUE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFETrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFETrap(EL3, TRUE);
      WaitForEvent();
  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFETrap(EL1, FALSE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFETrap(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFETrap(EL3, FALSE);
      WaitForInterrupt();
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    BTypeNext = '00';
  otherwise // do nothing
Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see *Wait For Interrupt*.

As described in *Wait For Interrupt*, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- **Traps to EL1 of EL0 execution of WFE and WFI instructions.**
- **Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions.**
- **Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions.**

```c
System

```SystemHintOp` op;

case CRm:op2 of
   when '0000 000' op = SystemHintOp_NOP;        // Instruction executes as NOP
   when '0000 001' op = SystemHintOp_YIELD;      // Instruction executes as NOP
   when '0000 010' op = SystemHintOp_WFE;        // Instruction executes as NOP
   when '0000 011' op = SystemHintOp_WFI;        // Instruction executes as NOP
   when '0000 100' op = SystemHintOp_SEV;        // Instruction executes as NOP
   when '0000 101' op = SystemHintOp_SEVL;       // Instruction executes as NOP
   when '0000 111'
       SEE "XPACLRI";
   when '0001 xxx'
       SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
   when '0010 000'
       if !HaveRASExt() then EndOfInstruction();    // Instruction executes as NOP
       op = SystemHintOp_ESB;
   when '0010 001'
       if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
       op = SystemHintOp_PSB;
   when '0010 010'
       if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
       op = SystemHintOp_TSB;
   when '0010 100'
       op = SystemHintOp_CSDB;
   when '0011 xxx'
       SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
   when '0100 xx0'
       op = SystemHintOp_BTI;
       BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);        // Instruction executes as NOP
   otherwise EndOfInstruction();        // Instruction executes as NOP
```

WFI
case op of
  when SystemHintOp YIELD
    Hint_Yield();
  when SystemHintOp WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, TRUE);
      end if
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, TRUE);
      end if
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, TRUE);
        WaitForEvent();
      end if
    end if
  when SystemHintOp WFI
    if !InterruptPending() then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, FALSE);
      end if
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, FALSE);
      end if
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, FALSE);
        WaitForInterrupt();
      end if
    end if
  when SystemHintOp SEV
    SendEvent();
  when SystemHintOp SEVL
    SendEventLocal();
  when SystemHintOp ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
      AArch64.vESBOperation();
      TakeUnmaskedSErrorInterrupts();
    end if
  when SystemHintOp PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp TSB
    TraceSynchronizationBarrier();
  when SystemHintOp CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp BTI
    BTypeNext = '00';
  otherwise // do nothing
XAFlag

Convert floating-point condition flags from external format to ARM format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an ARM floating-point scalar compare instruction.

System
(ARMv8.5)

```
1 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0 (0) (0) (0) 0 0 1 1 1 1 1
```

If !HaveFlagFormatExt() then UNDEFINED;

Operation

```
bit N = NOT(PSTATE.C) AND NOT(PSTATE.Z);
bit Z = PSTATE.Z AND PSTATE.C;
bit C = PSTATE.C OR PSTATE.Z;
bit V = NOT(PSTATE.C) AND PSTATE.Z;

PSTATE.N = N;
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = V;
```
Exclusive OR and Rotate performs a bitwise exclusive OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when ARMv8.2-SHA is implemented.

### Advanced SIMD (ARMv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | Rm |  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | Rm | imm6 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | Rn |  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 1  | 1  | 0  | 0  | 1  | 0  | 0  Rd |  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Advanced SIMD

XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>

if !HaveSHA3Ext () then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

### Assembler Symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<imm6>` is a rotation right, encoded in "imm6".

### Operation

```
AArch64.CheckFPAdvSIMDEnabled ();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) tmp;
tmp = Vn EOR Vm;
V[d] = ROR(tmp<127:64>, UInt(imm6));ROR(tmp<63:0>, UInt(imm6));
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for XPACI and XPACD, and is in LR for XPACLRI.

The XPACD instruction is used for data addresses, and XPACI and XPACLRI are used for instruction addresses.

It has encodings from 2 classes: Integer and System

### Integer (ARMv8.3)

```
1 1 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 Rd
```

#### XPACD (D == 1)

```
XPACD <Xd>
```

#### XPACI (D == 0)

```
XPACI <Xd>
```  
```java
boolean data = (D == '1');
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if n != 31 then UNDEFINED;
```

### System (ARMv8.3)

```
1 1 0 1 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1
```

### System

```
XPACLRI
```  
```java
integer d = 30;
boolean data = FALSE;
```

### Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

### Operation

```
if HavePACExt() then
    X[d] = Strip(X[d], data);
```
XTN, XTN2

Extract Narrow. This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The XTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the XTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |

**Vector**

\[
\text{XTN}\{2\} <Vd>.<Tb>, <Vn>.<Ta>
\]

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

| 11   |

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see The YIELD instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1 1
```

System

```
YIELD

 SystemHintOp op;

 case CRm:op2 of
   when '0000 000' op = SystemHintOp_NOP;
   when '0000 001' op = SystemHintOp_YIELD;
   when '0000 010' op = SystemHintOp_WFE;
   when '0000 011' op = SystemHintOp_WFI;
   when '0000 100' op = SystemHintOp_SEV;
   when '0000 101' op = SystemHintOp_SEVL;
   when '0000 111' SEE "XPACLRI";
   when '0001 xxx' SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
   when '0010 000'
     if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
     op = SystemHintOp_ESB;
   when '0010 001'
     if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
     op = SystemHintOp_PSB;
   when '0010 010'
     if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
     op = SystemHintOp_TSB;
   when '0010 100' op = SystemHintOp_CSDB;
   when '0011 xxx'
     SEE "PACIAZ, PACIAZP, PACIBZ, PACIBZP, AUTIAZ, AUTIAZP, AUTIBZ, AUTIBZP";
   when '0100 xx0' op = SystemHintOp_BTI;
     BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
   otherwise EndOfInstruction(); // Instruction executes as NOP
```
case op of
  when SystemHintOp_YIELD
      Hint_Yield();
  when SystemHintOp_WFE
      if IsEventRegisterSet() then
          ClearEventRegister();
      else
          if PSTATE.EL == EL0 then
              // Check for traps described by the OS which may be EL1 or EL2.
              AArch64.CheckForWFSympEl(EL1, TRUE);
          if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
              // Check for traps described by the Hypervisor.
              AArch64.CheckForWFSympEl(EL2, TRUE);
          if HaveEL(EL3) && PSTATE.EL != EL3 then
              // Check for traps described by the Secure Monitor.
              AArch64.CheckForWFSympEl(EL3, TRUE);
              WaitForEvent();
  when SystemHintOp_WFI
      if !InterruptPending() then
          if PSTATE.EL == EL0 then
              // Check for traps described by the OS which may be EL1 or EL2.
              AArch64.CheckForWFSympEl(EL1, FALSE);
          if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
              // Check for traps described by the Hypervisor.
              AArch64.CheckForWFSympEl(EL2, FALSE);
          if HaveEL(EL3) && PSTATE.EL != EL3 then
              // Check for traps described by the Secure Monitor.
              AArch64.CheckForWFSympEl(EL3, FALSE);
              WaitForInterrupt();
  when SystemHintOp_SEV
      SendEvent();
  when SystemHintOp_SEVL
      SendEventLocal();
  when SystemHintOp_ESB
      SynchronizeErrors();
      AArch64.ESBOperation();
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
      TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
      ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
      TraceSynchronizationBarrier();
  when SystemHintOp_CSD
      ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
      BTypeNext = '00';
  otherwise // do nothing
**ZIP1**

Zip vectors (primary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

This instruction can be used with **ZIP2** to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

---

**Advanced SIMD**

\[
\text{ZIP1 } <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
\]

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{ UInt }(Rn) \);
integer \( m = \text{UInt}(Rm) \);

\[
\text{if size:Q == '110' then UNDEFINED;\ninteger esize = 8 << \text{UInt}(size);\ninteger datasize = if Q == '1' then 128 else 64;\ninteger elements = datasize DIV esize;\ninteger part = \text{UInt}(op);\ninteger pairs = \text{elements DIV 2};\n}\]

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ZIP2

Zip vectors (secondary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

This instruction can be used with ZIP1 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

![Example Figure]

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

ZIP2 <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;

Assembler Symbols

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>  Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### Top-level encodings for A64

#### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100x</td>
<td>Data Processing -- Immediate</td>
</tr>
<tr>
<td>101x</td>
<td>Branches, Exception Generating and System instructions</td>
</tr>
<tr>
<td>x1x0</td>
<td>Loads and Stores</td>
</tr>
<tr>
<td>x101</td>
<td>Data Processing -- Register</td>
</tr>
<tr>
<td>x111</td>
<td>Data Processing -- Scalar Floating-Point and Advanced SIMD</td>
</tr>
</tbody>
</table>

#### Reserved

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>op0</th>
<th>0000</th>
<th>op1</th>
</tr>
</thead>
</table>

#### Data Processing -- Immediate

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td>PC-rel. addressing</td>
</tr>
</tbody>
</table>

#### PC-rel. addressing

These instructions are under Data Processing -- Immediate.

<table>
<thead>
<tr>
<th>op</th>
<th>immlo</th>
<th>immhi</th>
<th>Rd</th>
</tr>
</thead>
</table>
Add/subtract (immediate, with tags)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>ADR</td>
</tr>
<tr>
<td></td>
<td>ADRP</td>
</tr>
</tbody>
</table>

Add/subtract (immediate)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf op S</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0 0</td>
<td>ADDG</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>1 1 0</td>
<td>SUBG</td>
<td>ARMv8.5</td>
</tr>
</tbody>
</table>

Add/subtract (immediate)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf op S</td>
<td>ADD (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 32-bit</td>
</tr>
<tr>
<td>0 0 0</td>
<td>ADDS (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 32-bit</td>
</tr>
<tr>
<td>0 1 0</td>
<td>SUB (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 32-bit</td>
</tr>
<tr>
<td>0 1 1</td>
<td>SUBS (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 32-bit</td>
</tr>
<tr>
<td>1 0 0</td>
<td>ADD (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 64-bit</td>
</tr>
<tr>
<td>1 0 1</td>
<td>ADDS (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 64-bit</td>
</tr>
<tr>
<td>1 1 0</td>
<td>SUB (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 64-bit</td>
</tr>
<tr>
<td>1 1 1</td>
<td>SUBS (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 64-bit</td>
</tr>
</tbody>
</table>

Logical (immediate)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf opc N</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 0</td>
<td>AND (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 32-bit</td>
</tr>
<tr>
<td>0 0 1</td>
<td>ORR (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 32-bit</td>
</tr>
<tr>
<td>0 1 0</td>
<td>EOR (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 32-bit</td>
</tr>
<tr>
<td>0 11 0</td>
<td>ANDS (immediate)</td>
</tr>
<tr>
<td></td>
<td>— 32-bit</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>N</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>AND (immediate) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>ORR (immediate) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>EOR (immediate) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>ANDS (immediate) — 64-bit</td>
<td></td>
</tr>
</tbody>
</table>

### Move wide (immediate)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00</td>
<td>hw</td>
<td>imm16</td>
<td>Rd</td>
</tr>
</tbody>
</table>

### Bitfield

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>hw</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>hw</td>
<td>immr</td>
<td>imms</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

### Extract

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op21</th>
<th>N</th>
<th>o0</th>
<th>imms</th>
<th>Rm</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x1</td>
<td>N</td>
<td>o0</td>
<td>imms</td>
<td>Rm</td>
<td>imms</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>
**Instruction Details**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>0xxxxxxxxxxxxxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1xxxxxxxxxxxxxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>00xxxxxxxxxxxxx</td>
<td>Exception generation</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0100000000001x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0100000010001x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>01000000110010</td>
<td>Hints</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>01000000110010</td>
<td>!= 11111</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>01000000110011</td>
<td>Barriers</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>010000001xx001x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>01000000xxx000x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>01000000xxx0100</td>
<td>PSTATE</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>01000000xxx0101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>01000000xxx011x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>01000000xxx1xxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0100x01xxxxxxxxx</td>
<td>System instructions</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0100x1xxxxxxx</td>
<td>System register move</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0101xxxxxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>011xxxxxxxxxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1xxxxxxxxxxx</td>
<td>Unconditional branch (register)</td>
<td></td>
</tr>
<tr>
<td>x00</td>
<td></td>
<td>Unconditional branch (register)</td>
<td></td>
</tr>
<tr>
<td>x01</td>
<td>0xxxxxxxxxxxxxxx</td>
<td>Compare and branch (immediate)</td>
<td></td>
</tr>
<tr>
<td>x01</td>
<td>1xxxxxxxxxxxxxxx</td>
<td>Test and branch (immediate)</td>
<td></td>
</tr>
<tr>
<td>x11</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

**Conditional branch (immediate)**

These instructions are under Branches, Exception Generating and System instructions.
### Exception generation

These instructions are under **Branches, Exception Generating and System instructions**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>B.cond</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Hints

These instructions are under **Branches, Exception Generating and System instructions**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRm</td>
<td>HINT</td>
<td>-</td>
</tr>
<tr>
<td>0000 000</td>
<td>NOP</td>
<td>-</td>
</tr>
<tr>
<td>0000 001</td>
<td>YIELD</td>
<td>-</td>
</tr>
<tr>
<td>0000 010</td>
<td>WFE</td>
<td>-</td>
</tr>
<tr>
<td>0000 011</td>
<td>WFI</td>
<td>-</td>
</tr>
<tr>
<td>0000 100</td>
<td>SEV</td>
<td>-</td>
</tr>
<tr>
<td>0000 101</td>
<td>SEVL</td>
<td>-</td>
</tr>
</tbody>
</table>
### Top-level encodings for A64

#### Convertion fields

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>111</td>
<td>XPACD, XPACI, XPACLR</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0001</td>
<td>000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAXZ, PACIZA — PACIA1716</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0001</td>
<td>010</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB1716</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0001</td>
<td>100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIZA — AUTIA1716</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0001</td>
<td>110</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIZB — AUTIB1716</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0010</td>
<td>000</td>
<td>ESB</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0010</td>
<td>001</td>
<td>PSB CSYNC</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0010</td>
<td>010</td>
<td>TSB CSYNC</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0010</td>
<td>100</td>
<td>CSDB</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAXZ, PACIZA — PACIA</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0011</td>
<td>001</td>
<td>PACIA, PACIA1716, PACIASP, PACIAXZ, PACIZA — PACIASP</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0011</td>
<td>010</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBZ</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0011</td>
<td>011</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBSP</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0011</td>
<td>100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIZA — AUTIAZ</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0011</td>
<td>101</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIZA — AUTIASP</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0011</td>
<td>110</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIZB — AUTIBZ</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0011</td>
<td>111</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIZB — AUTIBSP</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0100</td>
<td>xx0</td>
<td>BTI</td>
<td>ARMv8.5</td>
</tr>
</tbody>
</table>

#### Barriers

These instructions are under Branches, Exception Generating and System instructions.

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Rt</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>010</td>
<td>111111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0001</td>
<td>011</td>
<td>111111</td>
<td>CLREX</td>
</tr>
<tr>
<td>0010</td>
<td>000</td>
<td>111111</td>
<td>DMN</td>
</tr>
<tr>
<td>0011</td>
<td>000</td>
<td>111111</td>
<td>ISB</td>
</tr>
<tr>
<td>0100</td>
<td>111111</td>
<td>SB</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>111111</td>
<td>DSB</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>111111</td>
<td>SSBB</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

#### PSTATE

These instructions are under Branches, Exception Generating and System instructions.

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Rt</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>100</td>
<td>111111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0001</td>
<td>011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>100</td>
<td>PSSBB</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
### System instructions

These instructions are under [Branches, Exception Generating and System instructions](#).  

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11111</td>
<td>MSR (immediate)</td>
<td>-</td>
</tr>
<tr>
<td>000 000</td>
<td>11111</td>
<td>CFINV ARMv8.4</td>
</tr>
<tr>
<td>000 001</td>
<td>11111</td>
<td>XAFlag ARMv8.5</td>
</tr>
<tr>
<td>000 010</td>
<td>11111</td>
<td>AXFlag ARMv8.5</td>
</tr>
</tbody>
</table>

### System register move

These instructions are under [Branches, Exception Generating and System instructions](#).  

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SYS</td>
</tr>
<tr>
<td>1</td>
<td>SYSL</td>
</tr>
</tbody>
</table>

### Unconditional branch (register)

These instructions are under [Branches, Exception Generating and System instructions](#).  

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MSR (register)</td>
</tr>
<tr>
<td>1</td>
<td>MRS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opc</th>
<th>op2</th>
<th>Decode fields</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 11111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0000 11111</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0000 11111</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>BR</td>
<td>-</td>
</tr>
<tr>
<td>0000 11111</td>
<td>000001</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0000 11111</td>
<td>000010</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0000 11111</td>
<td>000010</td>
<td>11111</td>
<td></td>
<td></td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key A, zero modifier</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0000 11111</td>
<td>000011</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0000 11111</td>
<td>000011</td>
<td>11111</td>
<td></td>
<td></td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key B, zero modifier</td>
<td>ARMv8.3</td>
</tr>
</tbody>
</table>
## Top-level encodings for A64

<table>
<thead>
<tr>
<th>opc</th>
<th>op2</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction Details</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>1111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>1111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>1111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>000000</td>
<td>BLR</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>000001</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>000010</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>000011</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, zero modifier</td>
<td>ARMv8.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>000011</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>000011</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier</td>
<td>ARMv8.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000000</td>
<td>RET</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000001</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000010</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000011</td>
<td>RETAA, RETAB — RETAA</td>
<td>ARMv8.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>1111</td>
<td>000001</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000000</td>
<td>ERET</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000001</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000010</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000010</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000010</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### Unconditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>op2</th>
<th>Decode fields</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>ERETTA, ERETAB — ERETTA</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000011</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000011</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>!=</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>11111</td>
<td>ERETTA, ERETAB — ERETAB</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>!=</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>!=</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>11111</td>
<td>!=</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>11111</td>
<td>00000</td>
<td>DRPS</td>
<td>-</td>
</tr>
<tr>
<td>0011x</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>0000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>00010</td>
<td>BRAAA, BRAAZ, BRAB, BRABZ — key A, register modifier</td>
<td>ARMv8.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>00011</td>
<td>BRAAA, BRAAZ, BRAB, BRABZ — key B, register modifier</td>
<td>ARMv8.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>01xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>000000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>000010</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, register modifier</td>
<td>ARMv8.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>000011</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, register modifier</td>
<td>ARMv8.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>01xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011x</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11xx</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Compare and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Test and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>b5</th>
<th>op</th>
<th>imm14</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Loads and Stores

These instructions are under the [top-level](#).

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
</tr>
</thead>
</table>

#### Instruction details

- `000000`: Advanced SIMD load/store multiple structures
- `010xxxxx`: Advanced SIMD load/store multiple structures (post-indexed)
- `0x1xxxxx`: UNALLOCATED
- `x000000`: Advanced SIMD load/store single structure
- `x1xxxxx`: Advanced SIMD load/store single structure (post-indexed)
- `x0xxxxx`: UNALLOCATED
- `x0xxxxx`: UNALLOCATED
- `x0xxxxx`: UNALLOCATED
- `x0xxxxx`: UNALLOCATED
- `1xxxxx`: UNALLOCATED
- `1xxxxx`: Load/store memory tags
- `1xxxxx`: UNALLOCATED

---

Top-level encodings for A64
### Advanced SIMD load/store multiple structures

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>Decode fields L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>0 0000</td>
<td>ST4 (multiple structures)</td>
</tr>
<tr>
<td>0 0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0010</td>
<td>ST1 (multiple structures) — four registers</td>
</tr>
<tr>
<td>0 0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0100</td>
<td>ST3 (multiple structures)</td>
</tr>
<tr>
<td>0 0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0110</td>
<td>ST1 (multiple structures) — three registers</td>
</tr>
<tr>
<td>0 0111</td>
<td>ST1 (multiple structures) — one register</td>
</tr>
<tr>
<td>0 1000</td>
<td>ST2 (multiple structures)</td>
</tr>
<tr>
<td>0 1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1010</td>
<td>ST1 (multiple structures) — two registers</td>
</tr>
<tr>
<td>0 1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 11xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0000</td>
<td>LD4 (multiple structures)</td>
</tr>
<tr>
<td>1 0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0010</td>
<td>LD1 (multiple structures) — four registers</td>
</tr>
<tr>
<td>1 0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0100</td>
<td>LD3 (multiple structures)</td>
</tr>
<tr>
<td>1 0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0110</td>
<td>LD1 (multiple structures) — three registers</td>
</tr>
<tr>
<td>1 0111</td>
<td>LD1 (multiple structures) — one register</td>
</tr>
<tr>
<td>1 1000</td>
<td>LD2 (multiple structures)</td>
</tr>
<tr>
<td>1 1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1010</td>
<td>LD1 (multiple structures) — two registers</td>
</tr>
</tbody>
</table>
## Advanced SIMD load/store multiple structures (post-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 11xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L</th>
<th>Rm</th>
<th>opcode</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L</th>
<th>Rm</th>
<th>opcode</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>!=</td>
<td>11111</td>
<td>0000</td>
<td>ST4 (multiple structures) — register offset</td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>11111</td>
<td>0010</td>
<td>ST1 (multiple structures) — four registers, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>11111</td>
<td>0100</td>
<td>ST3 (multiple structures) — register offset</td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>11111</td>
<td>0110</td>
<td>ST1 (multiple structures) — three registers, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>11111</td>
<td>0111</td>
<td>ST1 (multiple structures) — one register, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!=</td>
<td>11111</td>
<td>1000</td>
<td>ST2 (multiple structures) — register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>1010</td>
<td>ST1 (multiple structures) — two registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L</th>
<th>Rm</th>
<th>opcode</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>0000</td>
<td>LD4 (multiple structures) — register offset</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>0010</td>
<td>LD1 (multiple structures) — four registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>0100</td>
<td>LD3 (multiple structures) — register offset</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>0110</td>
<td>LD1 (multiple structures) — three registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>1011</td>
<td>LD1 (multiple structures) — one register, register offset</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>1000</td>
<td>LD2 (multiple structures) — register offset</td>
</tr>
<tr>
<td>1</td>
<td>!=</td>
<td>11111</td>
<td>1010</td>
<td>LD1 (multiple structures) — two registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>0000</td>
<td>LD4 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>0010</td>
<td>LD1 (multiple structures) — four registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>0100</td>
<td>LD3 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>0110</td>
<td>LD1 (multiple structures) — three registers, immediate offset</td>
</tr>
</tbody>
</table>
## Advanced SIMD load/store single structure

These instructions are under **Loads and Stores**.

### Table 1: Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>Rm</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11111</td>
<td>0111</td>
<td>LD1 (multiple structures) — one register, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>1000</td>
<td>LD2 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>1010</td>
<td>LD1 (multiple structures) — two registers, immediate offset</td>
</tr>
</tbody>
</table>

### Table 2: Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td></td>
<td></td>
<td>ST1 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td></td>
<td></td>
<td>ST3 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>ST1 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>ST3 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>00</td>
<td></td>
<td>ST1 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>01</td>
<td></td>
<td>ST1 (single structure) — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>00</td>
<td></td>
<td>ST3 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>01</td>
<td></td>
<td>ST3 (single structure) — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>000</td>
<td></td>
<td></td>
<td>ST2 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>001</td>
<td></td>
<td></td>
<td>ST4 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>ST2 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>ST4 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>00</td>
<td></td>
<td>ST2 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>01</td>
<td></td>
<td>ST2 (single structure) — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>00</td>
<td></td>
<td>ST4 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>01</td>
<td></td>
<td>ST4 (single structure) — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td></td>
<td></td>
<td>LD1 (single structure) — 8-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001</td>
<td></td>
<td></td>
<td>LD3 (single structure) — 8-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>LD1 (single structure) — 16-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### Advanced SIMD load/store single structure (post-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x0</td>
<td>16-bit</td>
<td>LD3 (single structure) — 16-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>00</td>
<td>32-bit</td>
<td>LD1 (single structure) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>00</td>
<td>64-bit</td>
<td>LD1 (single structure) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>01</td>
<td>LD3 (single structure) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>LD1R</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td>LD3R</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>000</td>
<td>00</td>
<td>8-bit</td>
<td>LD2 (single structure) — 8-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>x0</td>
<td>16-bit</td>
<td>LD2 (single structure) — 16-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>x0</td>
<td>16-bit</td>
<td>LD4 (single structure) — 16-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>00</td>
<td>32-bit</td>
<td>LD2 (single structure) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>01</td>
<td>64-bit</td>
<td>LD2 (single structure) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>00</td>
<td>LD4 (single structure) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>01</td>
<td>LD4 (single structure) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>0</td>
<td>LD2R</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>0</td>
<td>LD4R</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>Rm</td>
<td>opcode</td>
<td>S</td>
<td>size</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>--------</td>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>1</td>
<td>×1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>001</td>
<td>ST3 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>010</td>
<td>×0</td>
<td>ST1 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>011</td>
<td>×0</td>
<td>ST3 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td>ST1 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td>ST3 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>000</td>
<td>ST1 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>001</td>
<td>ST3 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>== 11111</td>
<td>010</td>
<td>×0</td>
<td>ST1 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>011</td>
<td>×0</td>
<td>ST3 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td>ST1 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td>ST3 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>×1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>×1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>×1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>×1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>000</td>
<td>ST2 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>001</td>
<td>ST4 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>010</td>
<td>×0</td>
<td>ST2 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>011</td>
<td>×0</td>
<td>ST4 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td>ST2 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td>ST4 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>000</td>
<td>ST2 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>001</td>
<td>ST4 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>010</td>
<td>×0</td>
<td>ST2 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>011</td>
<td>×0</td>
<td>ST4 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>ST2 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>ST4 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>×1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>×1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>Rm</td>
<td>opcode</td>
<td>S</td>
<td>size</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>----</td>
<td>--------</td>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>000</td>
<td>LD1 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>001</td>
<td>LD3 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>010</td>
<td>x0</td>
<td>LD1 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>011</td>
<td>x0</td>
<td>LD3 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>100</td>
<td>00</td>
<td>LD1 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>100</td>
<td>01</td>
<td>LD1 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>101</td>
<td>00</td>
<td>LD3 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>101</td>
<td>01</td>
<td>LD3 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>110</td>
<td>0</td>
<td>LD1R — register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>111</td>
<td>0</td>
<td>LD3R — register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>000</td>
<td>LD1 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>001</td>
<td>LD3 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>LD1 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>LD3 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>LD1 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td>LD3 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>LD3 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td>LD3 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>000</td>
<td>LD2 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>001</td>
<td>LD4 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>010</td>
<td>x0</td>
<td>LD2 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>011</td>
<td>x0</td>
<td>LD4 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>100</td>
<td>00</td>
<td>LD2 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>100</td>
<td>01</td>
<td>LD2 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>101</td>
<td>00</td>
<td>LD4 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>101</td>
<td>01</td>
<td>LD4 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>110</td>
<td>0</td>
<td>LD2R — register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>! = 11111</td>
<td>111</td>
<td>0</td>
<td>LD4R — register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>000</td>
<td>LD2 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
</tbody>
</table>
## Load/store memory tags

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>opc</th>
<th>imm9</th>
<th>op2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>01</td>
<td>STG — post-index</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>10</td>
<td>STG — signed offset</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>11</td>
<td>STG — pre-index</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>00</td>
<td>LDG</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>STZG — post-index</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>10</td>
<td>STZG — signed offset</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>11</td>
<td>STZG — pre-index</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>01</td>
<td>ST2G — post-index</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>ST2G — signed offset</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>ST2G — pre-index</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0000000000</td>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>01</td>
<td>STZ2G — post-index</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>10</td>
<td>STZ2G — signed offset</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>STZ2G — pre-index</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0000000000</td>
<td>00</td>
<td>LDGV</td>
<td>ARMv8.5</td>
</tr>
</tbody>
</table>

## Load/store exclusive

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>size</th>
<th>02</th>
<th>L</th>
<th>o1</th>
<th>o0</th>
<th>Rs</th>
<th>o0</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STXRB</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

---

Top-level encodings for A64
<table>
<thead>
<tr>
<th>size</th>
<th>o2</th>
<th>L</th>
<th>o1</th>
<th>o0</th>
<th>Rt2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STLXRB</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASP</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASPL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDXR</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDAXRB</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASPA</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>SPLL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>SPLL</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDDL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDDL</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASLB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STXR</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>STXR</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASP</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASPL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDXR</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDXR</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASH, CASAH, CASALH, CASLH — CASH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASH, CASAH, CASALH, CASLH — CASH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDLRH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDLRH</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASH, CASAH, CASALH, CASLH — CASA</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASH, CASAH, CASALH, CASLH — CASAL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>STXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>STXRP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>STXLP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>LDLXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>LDLXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STLR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CAS</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CAS</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDLAR — 32-bit</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDLAR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CAS</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CAS</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>size</td>
<td>o2</td>
<td>L</td>
<td>o1</td>
<td>o0</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----------------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STXR — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STLXR — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>STXP — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>STLXP — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDXR — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>LDAXR — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>LDXP — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>LDAXP — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STLLR — 64-bit</td>
<td>ARMv8.1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STLR — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111                  CAS, CASA, CASAL, CASL — 64-bit CAS</td>
<td>ARMv8.1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111                  CAS, CASA, CASAL, CASL — 64-bit CASL</td>
<td>ARMv8.1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1111                  LDARR — 64-bit</td>
<td>ARMv8.1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111                  LSDR — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1111                  CAS, CASA, CASAL, CASL — 64-bit CASA</td>
<td>ARMv8.1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1111                  CAS, CASA, CASAL, CASL — 64-bit CASAL</td>
<td>ARMv8.1</td>
<td></td>
</tr>
</tbody>
</table>

**Load register (literal)**

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>opc</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>V</th>
<th>0</th>
<th>0</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
</table>

**Load/store no-allocate pair (offset)**

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>opc</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>V</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>L</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>opc</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>STNP — 32-bit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDNP — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>STNP (SIMD&amp;FP) — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDNP (SIMD&amp;FP) — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>STNP (SIMD&amp;FP) — 64-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Load/store register pair (post-indexed)

These instructions are under Loads and Stores.

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>STGP</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDPSW</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>STP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Load/store register pair (offset)

These instructions are under Loads and Stores.

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>STGP</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDPSW</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>STP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 128-bit</td>
<td>-</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Load/store register pair (pre-indexed)

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Load/store register (unscaled immediate)

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>V</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>V</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STURB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>LDURB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDURB — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDURSB — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDURSB — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>STUR (SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>LDUR (SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STUR (SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDUR (SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>STURH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LDURH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDURSH — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDURSH — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>STUR (SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LDUR (SIMD&amp;FP) — 16-bit</td>
</tr>
</tbody>
</table>
### Load/store register (immediate post-indexed)

These instructions are under [Loads and Stores](#).

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STUR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDUR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDURSW</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STUR (SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDUR (SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STUR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDUR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>PRFM (unscaled offset)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STUR (SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDUR (SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>

### Instruction Details

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>
### Load/store register (unprivileged)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>imm9</th>
<th>opc</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STTRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDTRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDTRSB — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDTRSB — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STTRH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDTRH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDTRH — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDTRH — 32-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STTR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDTR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDTRSW</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STTR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDTR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Load/store register (immediate pre-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>imm9</th>
<th>opc</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
</tbody>
</table>

---

*Page 1625*
### Atomic memory operations

These instructions are under **Loads and Stores**.

![Memory Operations Table](image)

---

**Decode fields**

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
</tr>
</tbody>
</table>

---

**Instruction Details**

- **LDR (immediate)**: Load register.
- **LDRSW (immediate)**: Load register with stack pointer (SWP).
- **STR (immediate, SIMD&FP)**: Store register.

---

**Architecture Version**

- **ARMv8.1**
- **ARMv8.1**

---

**Page 1626**
<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>o3</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORAB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETAB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDMSAXB, LDMSAXAB, LDMSAXALB, LDMSAXLB — LDMSAXB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINAB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADDB, LDADDB, LDADDB, LDADDB — LDADDB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADDH, LDADDAH, LDADDAH, LDADDAH — LDADDAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRLH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORLH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRLH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDADDH, LDADDAH, LDADDAH, LDADDAH — LDADDAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORLH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRLH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPB</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>LDADDH, LDADDAH, LDADDAH, LDADDAH — LDADDAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRLH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORLH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>o3</td>
<td>opc</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>-----</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDAPRH</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADDDH, LDADDDAH, LDADDDALH, LDADDDLH — LDADDDAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRALH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORALH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETALH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINALH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPAH</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDDL — 32-bit LDADD</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWP</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDDL — 32-bit LDADD</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWP</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWPL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMIN</td>
<td>ARMv8.1</td>
</tr>
</tbody>
</table>
## Instruction Details

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>o3</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDAL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLR, LDCLR, LDCLR — 32-bit LDCLR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORAL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETAL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXAL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINAL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXAL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINAL</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLR, LDCLR, LDCLR — 64-bit LDCLR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWP</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>LDCLR, LDCLR, LDCLR, LDCLR — 64-bit LDCLR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLR, LDCLR, LDCLR — 64-bit LDCLR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWP</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>ARMv8.1</td>
</tr>
</tbody>
</table>
### Load/store register (register offset)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>STRB (register) — extended register</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>STRB (register) — shifted register</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>LDRB (register) — extended register</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>LDRB (register) — shifted register</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDRSB (register) — 64-bit with extended register offset</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDRSB (register) — 64-bit with shifted register offset</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>011</td>
<td>LDRSB (register) — 32-bit with extended register offset</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>STR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>STR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>LDR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STRH (register)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDRH (register)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>LDRSH (register) — 64-bit</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>LDRSH (register) — 32-bit</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STR (register) — 32-bit</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDR (register) — 32-bit</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>LDRSW (register)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STR (register) — 64-bit</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDR (register) — 64-bit</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>PRFM (register)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
<td>ARMv8.1</td>
</tr>
</tbody>
</table>
Load/store register (pac)

These instructions are under **Loads and Stores**.

```markdown
<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>V</th>
<th>0</th>
<th>0</th>
<th>M</th>
<th>S</th>
<th>1</th>
<th>imm9</th>
<th>W</th>
<th>1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>size M W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 0 0</td>
<td>LDRAA, LDRAB — key A, offset</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>11 0 1</td>
<td>LDRAA, LDRAB — key A, pre-indexed</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>11 1 0</td>
<td>LDRAA, LDRAB — key B, offset</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>11 1 1</td>
<td>LDRAA, LDRAB — key B, pre-indexed</td>
<td>ARMv8.3</td>
</tr>
</tbody>
</table>

Load/store register (unsigned immediate)

These instructions are under **Loads and Stores**.

```markdown
<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>V</th>
<th>0</th>
<th>1</th>
<th>opc</th>
<th>imm12</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size V opc</td>
<td></td>
</tr>
<tr>
<td>x1 1 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00 0 01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00 0 10</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00 0 11</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00 1 10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00 1 11</td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01 0 00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01 0 01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01 0 10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01 0 11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x 0 11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x 1 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 0 00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10 0 01</td>
<td>LDR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10 0 10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11 0 00</td>
<td>STR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11 0 01</td>
<td>LDR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11 0 10</td>
<td>PRFM (immediate)</td>
</tr>
<tr>
<td>11 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>
Data Processing -- Register

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0110</td>
<td></td>
<td>Data-processing (2 source)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0110</td>
<td></td>
<td>Data-processing (1 source)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0xxx</td>
<td></td>
<td>Logical (shifted register)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>xx00</td>
<td></td>
<td>Add-subtract (shifted register)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>xx11</td>
<td></td>
<td>Add-subtract (extended register)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>000000</td>
<td>Add/subtract (with carry)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>000011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>x00001</td>
<td>Rotate right into flags</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>xx00</td>
<td>010</td>
<td>Evaluate into flags</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>xxx0x</td>
<td>Conditional compare (register)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>xxx1x</td>
<td>Conditional compare (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0100</td>
<td></td>
<td>Conditional select</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>xx1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>xxx</td>
<td></td>
<td>Data-processing (3 source)</td>
</tr>
</tbody>
</table>

Data-processing (2 source)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>Rm</th>
<th>opcode</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0011xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>00001x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0000010</td>
<td>UDIV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>000011</td>
<td>SDIV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>000100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00011x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>001000</td>
<td>LSLV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>001001</td>
<td>LSRV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### Data-processing (1 source)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>opcode2</th>
<th>opcode</th>
<th>Rn</th>
<th>Rd</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>001010</td>
<td></td>
<td></td>
<td></td>
<td>ASRV — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001011</td>
<td></td>
<td></td>
<td></td>
<td>RORV — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010x11</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010000</td>
<td></td>
<td></td>
<td></td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010001</td>
<td></td>
<td></td>
<td></td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32H</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010010</td>
<td></td>
<td></td>
<td></td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32W</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010100</td>
<td></td>
<td></td>
<td></td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010101</td>
<td></td>
<td></td>
<td></td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010110</td>
<td></td>
<td></td>
<td></td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>SUBP</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000010</td>
<td></td>
<td></td>
<td></td>
<td>UDIV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000011</td>
<td></td>
<td></td>
<td></td>
<td>SDIV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td>LSVL — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td>LSRV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001010</td>
<td></td>
<td></td>
<td></td>
<td>ASRV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001100</td>
<td></td>
<td></td>
<td></td>
<td>PACGA</td>
<td>ARMv8.3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010xx0</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010x0x</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010011</td>
<td></td>
<td></td>
<td></td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010111</td>
<td></td>
<td></td>
<td></td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>SUBPS</td>
<td>ARMv8.5</td>
</tr>
</tbody>
</table>
Logical (shifted register)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>opcode</th>
<th>imm6</th>
<th>N</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x</td>
<td>UNALLOCATED</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>AND (shifted register)</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>ORR (shifted register)</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>ORN (shifted register)</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>EOR (shifted register)</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>EON (shifted register)</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>ANDS (shifted register)</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>BICS (shifted register)</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>AND (shifted register)</td>
<td>64-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>BIC (shifted register)</td>
<td>64-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>
### Add/subtract (shifted register)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>N</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>0</td>
<td></td>
<td>ORR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
<td></td>
<td>ORN (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
<td>EOR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>EON (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td></td>
<td>ANDS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td></td>
<td>BICS (shifted register) — 64-bit</td>
</tr>
</tbody>
</table>

### Add/subtract (extended register)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>S</th>
<th>shift</th>
<th>imm3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>xxxxx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>00</td>
<td>ADD (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>00</td>
<td>ADDS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>00</td>
<td>SUB (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>00</td>
<td>SUBS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>00</td>
<td>ADD (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>00</td>
<td>ADDS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>00</td>
<td>SUB (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>00</td>
<td>SUBS (shifted register) — 64-bit</td>
</tr>
</tbody>
</table>
Add/subtract (with carry)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>ADC — 32-bit</td>
</tr>
<tr>
<td>0 0 1</td>
<td>ADCS — 32-bit</td>
</tr>
<tr>
<td>0 1 0</td>
<td>SBC — 32-bit</td>
</tr>
<tr>
<td>0 1 1</td>
<td>SBCS — 32-bit</td>
</tr>
<tr>
<td>1 0 0</td>
<td>ADC — 64-bit</td>
</tr>
<tr>
<td>1 0 1</td>
<td>ADCS — 64-bit</td>
</tr>
<tr>
<td>1 1 0</td>
<td>SBC — 64-bit</td>
</tr>
<tr>
<td>1 1 1</td>
<td>SBCS — 64-bit</td>
</tr>
</tbody>
</table>

Rotate right into flags

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0 0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>RMIF</td>
<td>ARMv8.4</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Evaluate into flags

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 != 000000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 000000 0 != 1101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 000000 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 000000 0 0 1101</td>
<td>SETF8, SETF16 — SETF8</td>
<td>ARMv8.4</td>
</tr>
<tr>
<td>0 0 1 000000 1 0 1101</td>
<td>SETF8, SETF16 — SETF16</td>
<td>ARMv8.4</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Conditional compare (register)

These instructions are under Data Processing -- Register.
## Conditional compare (immediate)

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 CCMN (register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 CCMP (register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 CCMN (register) — 64-bit</td>
</tr>
</tbody>
</table>

## Conditional select

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 CCMP (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 CCMP (immediate) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 CCMN (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 CCMN (immediate) — 64-bit</td>
</tr>
</tbody>
</table>

## Data-processing (3 source)

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>op2</th>
<th>Rm</th>
<th>cond</th>
<th>op2</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

Top-level encodings for A64
### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>op54</th>
<th>op31</th>
<th>o0</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>010</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>011</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>100</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>110</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>111</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>000</td>
<td>0</td>
<td>MADD — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>000</td>
<td>1</td>
<td>MSUB — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>001</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>001</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>010</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>101</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>101</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>110</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>000</td>
<td>0</td>
<td>MADD — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>000</td>
<td>1</td>
<td>MSUB — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>001</td>
<td>0</td>
<td>SMADDL</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>001</td>
<td>1</td>
<td>SMSUBL</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>010</td>
<td>0</td>
<td>SMULH</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>101</td>
<td>0</td>
<td>UMADDL</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>101</td>
<td>1</td>
<td>UMSUBL</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>110</td>
<td>0</td>
<td>UMULH</td>
</tr>
</tbody>
</table>

### Data Processing -- Scalar Floating-Point and Advanced SIMD

These instructions are under the **top-level**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Decode fields</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
<th>Architecture version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x</td>
<td>x101</td>
<td>00xxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0x</td>
<td>x101</td>
<td>00xxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>0x</td>
<td>x101</td>
<td>00xxxx10</td>
<td>Cryptographic AES</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxx0xxx0</td>
<td>Cryptographic three-register SHA</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x101</td>
<td>00xxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0x</td>
<td>x101</td>
<td>00xxxx10</td>
<td>Cryptographic two-register SHA</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxx0xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0x</td>
<td>x101</td>
<td>00xxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>00</td>
<td>00xx</td>
<td>xxxx0xxx1</td>
<td>Advanced SIMD scalar copy</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>01</td>
<td>00xx</td>
<td>xxxx0xxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>0111</td>
<td>00xxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxxx0xxx1</td>
<td>Advanced SIMD scalar three same FP16</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxxx01xxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>1111</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar two-register miscellaneous FP16</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>------</td>
<td>-----------</td>
<td>-----------------------------------------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxxx1</td>
<td>Advanced SIMD scalar three same extra</td>
<td>ARMv8.1</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x100</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar two-register miscellaneous</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x110</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar pairwise</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>1xxxxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>x1xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx0</td>
<td>Advanced SIMD scalar three different</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx1</td>
<td>Advanced SIMD scalar three same</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>10</td>
<td>xxxxxx1</td>
<td></td>
<td>Advanced SIMD scalar shift by immediate</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>11</td>
<td>xxxxxxx1</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>1x</td>
<td>xxxxxxxx0</td>
<td></td>
<td>Advanced SIMD scalar x indexed element</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxxx0</td>
<td>Advanced SIMD table lookup</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxxx10</td>
<td>Advanced SIMD permute</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x0</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxxx0</td>
<td>Advanced SIMD extract</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>00</td>
<td>00xx</td>
<td>xxx0xxxxx1</td>
<td>Advanced SIMD copy</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>01</td>
<td>00xx</td>
<td>xxx0xxxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>0111</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>10xx</td>
<td>xxx0xxxxx10</td>
<td>Advanced SIMD three same (FP16)</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>10xx</td>
<td>xxx01xxxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>1111</td>
<td>00xxxxx10</td>
<td>Advanced SIMD two-register miscellaneous (FP16)</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxxx1</td>
<td>Advanced SIMD three same extra</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x100</td>
<td>00xxxxx10</td>
<td>Advanced SIMD two-register miscellaneous</td>
<td>ARMv8.5</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x110</td>
<td>00xxxxx10</td>
<td>Advanced SIMD across lanes</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x1xx</td>
<td>1xxxxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x1xx</td>
<td>x1xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx0</td>
<td>Advanced SIMD three different</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx1</td>
<td>Advanced SIMD three same</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>10</td>
<td>0000</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD modified immediate</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>10</td>
<td>!= 0000</td>
<td>xxxxxxx1</td>
<td>Advanced SIMD shift by immediate</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>11</td>
<td>xxxxxxxx1</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>1x</td>
<td>xxxxxxx0</td>
<td></td>
<td>Advanced SIMD vector x indexed element</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>00</td>
<td>10xx</td>
<td>xxx10xxxxx</td>
<td>Cryptographic three-register, imm2</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>00</td>
<td>11xx</td>
<td>xxx100xx</td>
<td>Cryptographic three-register SHA 512</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>00</td>
<td>xxx0xxxxx</td>
<td></td>
<td>Cryptographic four-register</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>01</td>
<td>00xx</td>
<td>xAR</td>
<td></td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>01</td>
<td>1000</td>
<td>0001000xx</td>
<td>Cryptographic two-register SHA 512</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>11x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1xx0</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x01x</td>
<td>0x</td>
<td>x0xx</td>
<td></td>
<td>Conversion between floating-point and fixed-point</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>x01x</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx000000</td>
<td>Conversion between floating-point and integer</td>
<td>ARMv8.3</td>
<td></td>
</tr>
<tr>
<td>x01x</td>
<td>0x</td>
<td>x1xx</td>
<td>x1000000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>x01x</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx10000</td>
<td>Floating-point data-processing (1 source)</td>
<td>ARMv8.5</td>
<td></td>
</tr>
<tr>
<td>x01x</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx100000</td>
<td>Floating-point compare</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>x01x</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx100000</td>
<td>Floating-point immediate</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>x01x</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx01</td>
<td>Floating-point conditional compare</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>x01x</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx10</td>
<td>Floating-point data-processing (2 source)</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>x01x</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx11</td>
<td>Floating-point conditional select</td>
<td>ARMv8.2</td>
<td></td>
</tr>
</tbody>
</table>
Cryptographic AES

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>000xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>lxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 00100</td>
<td>AESE</td>
</tr>
<tr>
<td>00 00101</td>
<td>AESD</td>
</tr>
<tr>
<td>00 00110</td>
<td>AESMC</td>
</tr>
<tr>
<td>00 00111</td>
<td>AESIMC</td>
</tr>
</tbody>
</table>

Cryptographic three-register SHA

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 000</td>
<td>SHA1C</td>
</tr>
<tr>
<td>00 001</td>
<td>SHA1P</td>
</tr>
<tr>
<td>00 010</td>
<td>SHA1M</td>
</tr>
<tr>
<td>00 011</td>
<td>SHA1SU0</td>
</tr>
<tr>
<td>00 100</td>
<td>SHA256H</td>
</tr>
<tr>
<td>00 101</td>
<td>SHA256H2</td>
</tr>
<tr>
<td>00 110</td>
<td>SHA256SU1</td>
</tr>
<tr>
<td>lx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Cryptographic two-register SHA

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxlxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>xlxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>lxxxx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### Advanced SIMD scalar copy

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

#### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>00000</td>
<td>SHA1H</td>
</tr>
<tr>
<td>00</td>
<td>00001</td>
<td>SHA1SU1</td>
</tr>
<tr>
<td>00</td>
<td>00010</td>
<td>SHA256SU0</td>
</tr>
<tr>
<td>00</td>
<td>00011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar three same FP16

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

#### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>imm5</th>
<th>imm4</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>xxx1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>xxx1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x1xx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>DUP</td>
<td>(element)</td>
</tr>
<tr>
<td>0</td>
<td>lxxx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x0000</td>
<td>0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>010xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1100x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>01111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11101</td>
<td>FCVTNS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>FCVTMS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11100</td>
<td>FCVTAS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11101</td>
<td>SCVT (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01100</td>
<td>FCMT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01101</td>
<td>FCMEQ (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01110</td>
<td>FCMLT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11010</td>
<td>FCVTPS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11011</td>
<td>FCVTZS (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11101</td>
<td>FRSQRTE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>FRECPE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>FRECPX</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11101</td>
<td>FCVTNU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>FCVTMU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11100</td>
<td>FCVTAU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01100</td>
<td>FCMEQ (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01101</td>
<td>FCMLE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11010</td>
<td>FCVTNU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11101</td>
<td>UNALOCATED</td>
</tr>
</tbody>
</table>

Advanced SIMD scalar three same extra

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>lxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
### Advanced SIMD scalar two-register miscellaneous

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0000</td>
<td>SQRDMLAH (vector)</td>
<td>ARMv8.1</td>
</tr>
<tr>
<td>1 0001</td>
<td>SQRDMLSH (vector)</td>
<td>ARMv8.1</td>
</tr>
</tbody>
</table>

Top-level encodings for A64
<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00011</td>
<td>USQADD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>SQNEG</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>CMGE (zero)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>CMLE (zero)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>NEG (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td>SQXTUN, SQXTUN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UQXTN, UQXTN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>10110</td>
<td>FCVTXN, FCVTXN2</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11010</td>
<td>FCVTNU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11100</td>
<td>FCVTAU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01100</td>
<td>FCMGE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01101</td>
<td>FCMLE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01110</td>
<td>FCVTPU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11011</td>
<td>FCVTZU (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td>FRSQRT</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**Advanced SIMD scalar pairwise**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>010xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11010</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1x 01110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11011</td>
<td>ADDP (scalar)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 00</td>
<td>01100</td>
<td>FMAXNMP (scalar) — half-precision</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0 00</td>
<td>01101</td>
<td>FADDP (scalar) — half-precision</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0 00</td>
<td>01111</td>
<td>FMAXP (scalar) — half-precision</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>01100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>01101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>01111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>01100</td>
<td>FMINNMP (scalar) — half-precision</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>01111</td>
<td>FMINP (scalar) — half-precision</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0 11</td>
<td>01100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 11</td>
<td>01111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD scalar three different

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11011</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01100</td>
<td>FMAXNMP (scalar) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01101</td>
<td>FADDP (scalar) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01111</td>
<td>FMAXP (scalar) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01100</td>
<td>FMINNMP (scalar) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01111</td>
<td>FMINP (scalar) — single-precision and double-precision</td>
<td>-</td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar three same

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1001</td>
<td>SQDMLAL, SQDMLAL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>SQDMLSL, SQDMLSL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>SQDMULL, SQDMULL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Decode fields size</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---</td>
<td>---------------------</td>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>SRSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>SQRSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10000</td>
<td>ADD (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10001</td>
<td>CMTST</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10110</td>
<td>SQDMULH (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11011</td>
<td>FMULX</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11100</td>
<td>FCMEQ (register)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11110</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11111</td>
<td>FRECPS</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11110</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11111</td>
<td>FRSQRTS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000001</td>
<td>UQADD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00101</td>
<td>UQSUB</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>CMHI (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>CMHS (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>USHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>UOSH (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>URSHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>UQRSHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10000</td>
<td>SUB (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10001</td>
<td>CMGE (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10110</td>
<td>SQDMULH (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11100</td>
<td>FCMEQ (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11101</td>
<td>FACGE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11110</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD scalar shift by immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Descode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1x</td>
<td>11010</td>
<td>FABD</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11100</td>
<td>FCMGT (register)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td>FACGT</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Top-level encodings for A64

---

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>00001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>00011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>00101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>00111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>01001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>01011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>01101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>101xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>110xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>11101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>== 0000</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>00000</td>
<td>SSHR</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>00010</td>
<td>SSRA</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>00100</td>
<td>SRSR</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>00110</td>
<td>SRSRA</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>01000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>01010</td>
<td>SHL</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>01100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>01110</td>
<td>SOSHL (immediate)</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>10000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>10001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>10010</td>
<td>SOSHRN, SOSHRN2</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>10011</td>
<td>SORSHRN, SORSHRN2</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>11100</td>
<td>SCVT (vector, fixed-point)</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>11111</td>
<td>FCVTZS (vector, fixed-point)</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>00000</td>
<td>USHR</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>00010</td>
<td>USRA</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>00100</td>
<td>USRSHR</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>00110</td>
<td>URSRA</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>01000</td>
<td>SRI</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>01010</td>
<td>SLI</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>01100</td>
<td>SOSHLU</td>
</tr>
</tbody>
</table>
### Advanced SIMD scalar x indexed element

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>01110</td>
<td>UQSHL (Immediate)</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>10000</td>
<td>SQSHRUN, SQSHRUN2</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>10001</td>
<td>SQRSHRUN, SQRSHRUN2</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>10010</td>
<td>UQSHRN, UQSHRN2</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>10011</td>
<td>UQSRHNR, UQSRHNR2</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>11100</td>
<td>UCVT (vector, fixed-point)</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>11111</td>
<td>FCVTZU (vector, fixed-point)</td>
</tr>
</tbody>
</table>

### Top-level encodings for A64

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 0101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0001</td>
<td>SQDMLAL, SQDMLAL2 (by element)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0111</td>
<td>SQDMLSL, SQDMLSL2 (by element)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1011</td>
<td>SQDMULL, SQDMULL2 (by element)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1100</td>
<td>SQDMULH (by element)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1101</td>
<td>SQRDMLULH (by element)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0001</td>
<td>FMUL (by element) — half-precision</td>
<td>ARMv8.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0001</td>
<td>FMU (by element) — single-precision and double-precision</td>
<td>ARMv8.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x 0001</td>
<td>FMUL (by element) — single-precision and double-precision</td>
<td>ARMv8.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0101</td>
<td>FMULX (by element) — half-precision</td>
<td>ARMv8.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1101</td>
<td>SQRDMLULH (by element)</td>
<td>ARMv8.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1111</td>
<td>SQRDMLSLH (by element)</td>
<td>ARMv8.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0001</td>
<td>FMULX (by element) — half-precision</td>
<td>ARMv8.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1x 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD Table Lookup

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1x</td>
<td>0101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1001</td>
<td>FMULX (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
</tbody>
</table>

### Advanced SIMD Permute

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>op2</th>
<th>len</th>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>TBL — single register table</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>TBX — single register table</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>0</td>
<td>TBL — two register table</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>1</td>
<td>TBX — two register table</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0</td>
<td>TBL — three register table</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>TBX — three register table</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>0</td>
<td>TBL — four register table</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>1</td>
<td>TBX — four register table</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD Extract

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>UZP1</td>
</tr>
<tr>
<td>010</td>
<td>TRN1</td>
</tr>
<tr>
<td>011</td>
<td>ZIP1</td>
</tr>
<tr>
<td>100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101</td>
<td>UZP2</td>
</tr>
<tr>
<td>110</td>
<td>TRN2</td>
</tr>
<tr>
<td>111</td>
<td>ZIP2</td>
</tr>
</tbody>
</table>

---

[Data Processing -- Scalar Floating-Point and Advanced SIMD](#)
Advanced SIMD copy

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

Advanced SIMD three same (FP16)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 110</td>
<td>FMIN (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 1 111</td>
<td>FRSQRTS</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 0 000</td>
<td>FMAXNMP (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 0 001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0 010</td>
<td>FADDP (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 0 011</td>
<td>FMUL (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 0 100</td>
<td>FCMGE (register)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 0 101</td>
<td>FACGE</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 0 110</td>
<td>FMAXP (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 0 111</td>
<td>FDIV (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 1 000</td>
<td>FMINNMP (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 1 001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1 010</td>
<td>FABD</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 1 011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1 100</td>
<td>FCMGT (register)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 1 101</td>
<td>FACGT</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 1 110</td>
<td>FMINP (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1 1 111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

**Advanced SIMD two-register miscellaneous (FP16)**

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD.**

---

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 011xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 11100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 11000</td>
<td>FRINTN (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 0 11001</td>
<td>FRINTM (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 0 11010</td>
<td>FCVTNS (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 0 11011</td>
<td>FCVTMS (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 0 11100</td>
<td>FCVTAS (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 0 11101</td>
<td>SCVT (vector, integer)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 1 01100</td>
<td>FCMGT (zero)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 1 01101</td>
<td>FCMEQ (zero)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 1 01110</td>
<td>FCMLT (zero)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 1 01111</td>
<td>FABS (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 1 11000</td>
<td>FRINTP (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 1 11001</td>
<td>FRINTZ (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 1 11010</td>
<td>FCVTPS (vector)</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0 1 11011</td>
<td>FCVTZS (vector, integer)</td>
<td>ARMv8.2</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>11101</td>
<td>FRECPE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11000</td>
<td>FRINTA (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11001</td>
<td>FRINTX (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11010</td>
<td>FCVTN (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11100</td>
<td>FCVTU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01100</td>
<td>FCMGE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01101</td>
<td>FCMLE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01111</td>
<td>FNEG (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11001</td>
<td>FRINTI (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11010</td>
<td>FCVT (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11011</td>
<td>FCVTZU (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11101</td>
<td>FRSTQ (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>FSQRT (vector)</td>
</tr>
</tbody>
</table>

### Advanced SIMD three same extra

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

### Advanced SIMD two-register miscellaneous

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.
<table>
<thead>
<tr>
<th>U size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>011xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>10110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>11110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00000</td>
<td>REV64</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00001</td>
<td>REV16 (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00010</td>
<td>SADLP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00011</td>
<td>SUOADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00100</td>
<td>CLS (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00101</td>
<td>CNT</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>SADLP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>SQABS</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>CMGT (zero)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01001</td>
<td>CMEQ (zero)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>CMLT (zero)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>ABS</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10010</td>
<td>XTN, XTN2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>SQXTN, SQXTN2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>10110</td>
<td>FCVTN, FCVTN2</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>10111</td>
<td>FCVTL, FCVTL2</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11000</td>
<td>FRINTN (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11001</td>
<td>FRINTM (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11010</td>
<td>FCVTNS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11011</td>
<td>FCVTMS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11100</td>
<td>FCVTAS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11101</td>
<td>SCVTF (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11110</td>
<td>FRINT32Z (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11111</td>
<td>FRINT64Z (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01100</td>
<td>FCMGT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01101</td>
<td>FCMEQ (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01110</td>
<td>FCMLT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01111</td>
<td>FABS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11000</td>
<td>FRINTP (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11001</td>
<td>FRINTZ (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11010</td>
<td>FCVTPS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11011</td>
<td>FCVTZS (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11100</td>
<td>URECPE</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11101</td>
<td>FRECPE</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>00000</td>
<td>REV32 (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00010</td>
<td>UADDLP</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00011</td>
<td>USQADD</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00100</td>
<td>CLZ (vector)</td>
<td>-</td>
</tr>
<tr>
<td>U</td>
<td>Decode fields size</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---</td>
<td>-------------------</td>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td></td>
<td>UADALP</td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td></td>
<td>SQNEG</td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td></td>
<td>CMGE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td></td>
<td>CMLE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td></td>
<td>NEG (vector)</td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td></td>
<td>SQXTUN, SQXTUN2</td>
</tr>
<tr>
<td>1</td>
<td>10011</td>
<td></td>
<td>SHLL, SHLL2</td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td></td>
<td>UQXTN, UQXTN2</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>10110</td>
<td>FCVTXN, FCVTXN2</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>10111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11000</td>
<td>FRINTA (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11001</td>
<td>FRINTX (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11010</td>
<td>FCVTNU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11100</td>
<td>FCVTAU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11110</td>
<td>FRINT32X (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11111</td>
<td>FRINT64X (vector)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00101</td>
<td>NOT</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00101</td>
<td>RBIT (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>00101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01100</td>
<td>FCMGE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01101</td>
<td>FCMLE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01111</td>
<td>FNEG (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11001</td>
<td>FRINTI (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11010</td>
<td>FCVTPU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11011</td>
<td>FCVTZU (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11100</td>
<td>URSQRTEN</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td>FRSQRTEN</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td>FSORT (vector)</td>
</tr>
</tbody>
</table>

**Advanced SIMD across lanes**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0 1 1 1 1 0</td>
<td>size</td>
<td>1 1 0 0 0</td>
</tr>
<tr>
<td>0000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

Advanced SIMD three different
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1110</td>
<td>PMULL, PMULL2</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>UADDL, UADDL2</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>UADDW, UADDW2</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>USUBL, USUBL2</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>USUBW, USUBW2</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>RADDHN, RADDHN2</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>UABAL, UABAL2</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>RSUBHN, RSUBHN2</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
<td>UABDL, UABDL2</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>UMLAL, UMLAL2 (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>UMLSL, UMLSL2 (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1100</td>
<td>UMULL, UMULL2 (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### Advanced SIMD three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>SHADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00001</td>
<td>SQADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00010</td>
<td>SRHADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00100</td>
<td>SHSUB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00101</td>
<td>SQSUB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>CMGT (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>CMGE (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>SSHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01001</td>
<td>SQSHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>SRSHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>SQRSHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01100</td>
<td>SMAX</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01101</td>
<td>SMIN</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01110</td>
<td>SABD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01111</td>
<td>SABA</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10000</td>
<td>ADD (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10001</td>
<td>CMTST</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10010</td>
<td>MLA (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10011</td>
<td>MUL (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>SMAXP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10101</td>
<td>SMINP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10110</td>
<td>SQDMULH (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10111</td>
<td>ADDP (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Decode fields</td>
<td>opcode</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>---</td>
<td>---------------</td>
<td>--------</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>0</td>
<td>0x 11000</td>
<td>FMAXNM (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11001</td>
<td>FMLA (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11010</td>
<td>FADD (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11011</td>
<td>FMULX</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11100</td>
<td>FCMEQ (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11110</td>
<td>FMAX (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11111</td>
<td>FRECPS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 00011</td>
<td>AND (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 11101</td>
<td>FMLAL, FMLAL2 (vector) — FMLAL</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01 00011</td>
<td>BIC (vector, register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01 11101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11000</td>
<td>FMINNM (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11001</td>
<td>FMLS (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11010</td>
<td>FSUB (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11110</td>
<td>FMIN (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11111</td>
<td>FRSQRTS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 00011</td>
<td>QRR (vector, register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 11101</td>
<td>FMLSL, FMLSL2 (vector) — FMLSL</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 00011</td>
<td>ORN (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 11101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000</td>
<td>UHADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UQADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00010</td>
<td>URHADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00100</td>
<td>UHSUB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00101</td>
<td>UQSUB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>CMHI (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>CMHS (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>USHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>UQSHL</td>
<td>(register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>URSHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>UQRSHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01100</td>
<td>UMAX</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01101</td>
<td>UMIN</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01110</td>
<td>UABD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01111</td>
<td>UABA</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10000</td>
<td>SUB (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10001</td>
<td>CMEQ (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td>MLS (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10011</td>
<td>PMUL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UMAXP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10101</td>
<td>UMINP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10110</td>
<td>SQRDMULH (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11000</td>
<td>FMAXNMP (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11101</td>
<td>FADDP (vector)</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD modified immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Q</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x0</td>
<td>MOV1 — 32-bit shifted immediate</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x1</td>
<td>ORR (vector, immediate) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10x0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10x1</td>
<td>ORR (vector, immediate) — 16-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>110x</td>
<td>MOV1 — 32-bit shifting ones</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>110x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>MOV1 — 8-bit</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>FMOV (vector, immediate) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>FMOV (vector, immediate) — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>10x0</td>
<td>MVNI — 32-bit shifted immediate</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10x1</td>
<td>BIC (vector, immediate) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10x0</td>
<td>MVNI — 16-bit shifted immediate</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10x1</td>
<td>BIC (vector, immediate) — 16-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>110x</td>
<td>MVNI — 32-bit shifting ones</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>MOV1 — 64-bit scalar</td>
<td>-</td>
</tr>
</tbody>
</table>
### Advanced SIMD shift by immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

The following constraints also apply to this encoding: immh != 0000 && immh != 0000
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 01010</td>
<td>SLI</td>
</tr>
<tr>
<td>U 01100</td>
<td>SQSHLU</td>
</tr>
<tr>
<td>U 01110</td>
<td>UQSHL (immediate)</td>
</tr>
<tr>
<td>U 10000</td>
<td>SQSHRUN, SQSHRUN2</td>
</tr>
<tr>
<td>U 10001</td>
<td>SQSHRUN, SQSHRUN2</td>
</tr>
<tr>
<td>U 10010</td>
<td>UQSHRN, UQSHRN2</td>
</tr>
<tr>
<td>U 10011</td>
<td>UQSHRN, UQSHRN2</td>
</tr>
<tr>
<td>U 10100</td>
<td>USHL, USHL2</td>
</tr>
<tr>
<td>U 11100</td>
<td>UCVTF (vector, fixed-point)</td>
</tr>
<tr>
<td>U 11111</td>
<td>FCVTZU (vector, fixed-point)</td>
</tr>
</tbody>
</table>

**Advanced SIMD vector x indexed element**

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0010</td>
<td>SMLAL, SMLAL2 (by element)</td>
</tr>
<tr>
<td>0 0011</td>
<td>SODMLAL, SODMLAL2 (by element)</td>
</tr>
<tr>
<td>0 0110</td>
<td>SMLSL, SMLSL2 (by element)</td>
</tr>
<tr>
<td>0 0111</td>
<td>SODMLSL, SODMLSL2 (by element)</td>
</tr>
<tr>
<td>0 1000</td>
<td>MUL (by element)</td>
</tr>
<tr>
<td>0 1010</td>
<td>SMULL, SMULL2 (by element)</td>
</tr>
<tr>
<td>0 1011</td>
<td>SQDMULL, SQDMULL2 (by element)</td>
</tr>
<tr>
<td>0 1100</td>
<td>SQDMULH (by element)</td>
</tr>
<tr>
<td>0 1101</td>
<td>SQDVMULH (by element)</td>
</tr>
<tr>
<td>0 1110</td>
<td>SDOT (by element)</td>
</tr>
<tr>
<td>0 1111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x 0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x 0100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0001</td>
<td>FMLA (by element) — half-precision</td>
</tr>
<tr>
<td>0 0101</td>
<td>FMLS (by element) — half-precision</td>
</tr>
<tr>
<td>0 1001</td>
<td>FMUL (by element) — half-precision</td>
</tr>
<tr>
<td>0 0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1x 0001</td>
<td>FMLA (by element) — single-precision and double-precision</td>
</tr>
<tr>
<td>0 1x 0101</td>
<td>FMLS (by element) — single-precision and double-precision</td>
</tr>
<tr>
<td>0 1x 1001</td>
<td>FMUL (by element) — single-precision and double-precision</td>
</tr>
<tr>
<td>0 10 0000</td>
<td>FMLAL, FMLAL2 (by element) — FMLAL</td>
</tr>
<tr>
<td>0 10 1000</td>
<td>FMLSL, FMLSL2 (by element) — FMLSL</td>
</tr>
<tr>
<td>0 11 0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0100</td>
<td>UMLA, UMLA2 (by element)</td>
</tr>
<tr>
<td>1 0100</td>
<td>MLS (by element)</td>
</tr>
<tr>
<td>U</td>
<td>size</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

**Cryptographic three-register, imm2**

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 1  | 0  | imm2 | opcode | Rn | Rd |

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SM3TT1A</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>01</td>
<td>SM3TT1B</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>10</td>
<td>SM3TT2A</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>11</td>
<td>SM3TT2B</td>
<td>ARMv8.2</td>
</tr>
</tbody>
</table>

**Cryptographic three-register SHA 512**

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | Rm | 1  | 0  | 0  | opcode | Rn | Rd |

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SHA512H</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>01</td>
<td>SHA512H2</td>
<td>ARMv8.2</td>
</tr>
</tbody>
</table>
### Cryptographic four-register

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SHA512SU1</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>RAX1</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>SM3PARTW1</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>SM3PARTW2</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>SM4EKEY</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Cryptographic two-register SHA 512

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>EOR3</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>01</td>
<td>BCAX</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>10</td>
<td>SM3SS1</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Conversion between floating-point and fixed-point

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>type</th>
<th>Decode fields</th>
<th>opcode</th>
<th>scale</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>lxx</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x0</td>
<td>00x</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x1</td>
<td>01x</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x</td>
<td>00x</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lx</td>
<td>01x</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>000</td>
<td>010</td>
<td>SCVT (scalar, fixed-point) — 32-bit to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>sf</td>
<td>S</td>
<td>Decode fields</td>
<td>D</td>
<td>m</td>
<td>n</td>
<td>D.</td>
<td>n.</td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>--------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>00</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>00</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>00</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>00</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>00</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>00</td>
<td>01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Conversion between floating-point and integer**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 001</td>
<td>FCVTNU (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 010</td>
<td>SCVT (scalar, integer) — 32-bit to single-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 011</td>
<td>UCVT (scalar, integer) — 32-bit to single-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 100</td>
<td>FCVTAS (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 101</td>
<td>FCVTAU (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 110</td>
<td>FMOV (general) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 111</td>
<td>FMOV (general) — 32-bit to single-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 01 000</td>
<td>FCVTPS (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 01 001</td>
<td>FCVTPU (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 11x 11x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 10 000</td>
<td>FCVTMS (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 10 001</td>
<td>FCVTMU (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 11 000</td>
<td>FCVTZS (scalar, integer) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 11 001</td>
<td>FCVTZU (scalar, integer) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 000</td>
<td>FCVTNS (scalar) — double-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 001</td>
<td>FCVTNU (scalar) — double-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 010</td>
<td>SCVT (scalar, integer) — 32-bit to double-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 011</td>
<td>UCVT (scalar, integer) — 32-bit to double-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 000</td>
<td>FCVTMS (scalar) — double-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 001</td>
<td>FCVTMU (scalar) — double-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 000</td>
<td>FCVTZS (scalar, integer) — double-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 001</td>
<td>FCVTZU (scalar, integer) — double-precision to 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 110</td>
<td>FJCVTZS</td>
<td>ARMv8.3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10 11x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 000</td>
<td>FCVTNS (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 001</td>
<td>FCVTNU (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 010</td>
<td>SCVT (scalar, integer) — 32-bit to half-precision</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 011</td>
<td>UCVT (scalar, integer) — 32-bit to half-precision</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 100</td>
<td>FCVTAS (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 101</td>
<td>FCVTAU (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 110</td>
<td>FMOV (general) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 111</td>
<td>FMOV (general) — 32-bit to half-precision</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 01 000</td>
<td>FCVTPS (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 01 001</td>
<td>FCVTPU (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 10 000</td>
<td>FCVTMS (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 10 001</td>
<td>FCVTMU (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 11 000</td>
<td>FCVTZS (scalar, integer) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 11 001</td>
<td>FCVTZU (scalar, integer) — half-precision to 32-bit</td>
<td>ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 11x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 000</td>
<td>FCVTNS (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>sf</td>
<td>S</td>
<td>Decode fields</td>
<td>rmode</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>---------------</td>
<td>-------</td>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00 001</td>
<td>FCVTNU (scalar) — single-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00 010</td>
<td>SCVTF (scalar, integer) — 64-bit to single-precision</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00 011</td>
<td>UCVTF (scalar, integer) — 64-bit to single-precision</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00 100</td>
<td>FCVTAS (scalar) — single-precision to 64-bit</td>
<td>101</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00 101</td>
<td>FCVTAU (scalar) — single-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 01 00 000</td>
<td>FCVTPS (scalar) — single-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 01 00 001</td>
<td>FCVTU (scalar) — single-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 10 00 000</td>
<td>FCVTMS (scalar) — single-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 10 00 001</td>
<td>FCVTMU (scalar) — single-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 11 00 000</td>
<td>FCVTZS (scalar, integer) — single-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 11 00 001</td>
<td>FCVTZU (scalar, integer) — single-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 x1 11x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 00 000</td>
<td>FCVTNS (scalar) — double-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 00 001</td>
<td>FCVTNU (scalar) — double-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 00 010</td>
<td>SCVTF (scalar, integer) — 64-bit to double-precision</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 00 011</td>
<td>UCVTF (scalar, integer) — 64-bit to double-precision</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 10 000</td>
<td>FCVTAS (scalar) — double-precision to 64-bit</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 10 001</td>
<td>FCVTAU (scalar) — double-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 11 000</td>
<td>FMOV (general) — double-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 00 11 001</td>
<td>FMOV (general) — double-precision to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 10 x0 11x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 10 01 110</td>
<td>FMOV (general) — top half of 128-bit to 64-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 10 01 111</td>
<td>FMOV (general) — 64-bit to top half of 128-bit</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01 10 1x 11x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 00 000</td>
<td>FCVTNS (scalar) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 00 001</td>
<td>FCVTNU (scalar) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 00 010</td>
<td>SCVTF (scalar, integer) — 64-bit to half-precision</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 00 011</td>
<td>UCVTF (scalar, integer) — 64-bit to half-precision</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 10 000</td>
<td>FCVTAS (scalar) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 10 001</td>
<td>FCVTAU (scalar) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 10 110</td>
<td>FMOV (general) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 10 111</td>
<td>FMOV (general) — 64-bit to half-precision</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 00 000</td>
<td>FCVTPS (scalar) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 00 001</td>
<td>FCVTU (scalar) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 01 000</td>
<td>FCVTMS (scalar) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 01 001</td>
<td>FCVTMU (scalar) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 10 00 000</td>
<td>FCVTZS (scalar, integer) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 10 00 001</td>
<td>FCVTZU (scalar, integer) — half-precision to 64-bit</td>
<td>0</td>
<td>ARMv8.2</td>
</tr>
</tbody>
</table>
## Floating-point data-processing (1 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>type</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000000</td>
<td>FMOV (register) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000001</td>
<td>FABS (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000010</td>
<td>FNEG (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000011</td>
<td>FSQRT (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000101</td>
<td>FCVT — single-precision to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000111</td>
<td>FCVT — single-precision to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>010000</td>
<td>FRINTN (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>010001</td>
<td>FRINTP (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>010100</td>
<td>FRINTM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>010110</td>
<td>FRINTZ (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>011000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>011100</td>
<td>FRINTX (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>011110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001000</td>
<td>FRINT32Z (scalar) — single-precision</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001001</td>
<td>FRINT32X (scalar) — single-precision</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001010</td>
<td>FRINT64Z (scalar) — single-precision</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001011</td>
<td>FRINT64X (scalar) — single-precision</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0101xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000000</td>
<td>FMOV (register) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000001</td>
<td>FABS (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000010</td>
<td>FNEG (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000011</td>
<td>FSQRT (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001000</td>
<td>FRINTN (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001001</td>
<td>FRINTP (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001010</td>
<td>FRINTM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001100</td>
<td>FRINTZ (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001110</td>
<td>FCVT — double-precision to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001111</td>
<td>FCVT — double-precision to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001001</td>
<td>FRINT32Z (scalar) — double-precision</td>
<td>ARMv8.5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001010</td>
<td>FRINT32X (scalar) — double-precision</td>
<td>ARMv8.5</td>
</tr>
</tbody>
</table>
### Floating-point compare

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>01 0100000</td>
<td>FCMPE</td>
<td>half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0101000</td>
<td>FCMPE</td>
<td>half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0110000</td>
<td>FCMPE</td>
<td>half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0111000</td>
<td>FCMPE</td>
<td>half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0111100</td>
<td>FCMPE</td>
<td>half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0111110</td>
<td>FCMPE</td>
<td>half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0111111</td>
<td>FCMPE</td>
<td>half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0111111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Floating-point immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>type</th>
<th>op</th>
<th>opcode2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>10000</td>
<td>FCMPE</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>11000</td>
<td>FCMPE</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

### Floating-point conditional compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>type</th>
<th>imm5</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0000</td>
<td>FMOV (scalar, immediate) single-precision</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0000</td>
<td>FMOV (scalar, immediate) double-precision</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0000</td>
<td>FMOV (scalar, immediate) half-precision ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

### Floating-point data-processing (2 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>type</th>
<th>op</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>FCCMP single-precision</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>FCCMP double-precision</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>FCCMP double-precision</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>FCCMP half-precision ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>FCCMP half-precision ARMv8.2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
### Floating-point conditional select

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>type</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>FMUL (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>FDIV (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0010</td>
<td>FADD (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0011</td>
<td>FSUB (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0100</td>
<td>FMAX (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0101</td>
<td>FMIN (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0110</td>
<td>FMAXNM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0111</td>
<td>FMINNM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>FMUL (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1001</td>
<td>FADD (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1010</td>
<td>FSUB (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1011</td>
<td>FMAX (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1100</td>
<td>FMIN (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>FMAXNM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1110</td>
<td>FMINNM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1111</td>
<td>FMUL (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0000</td>
<td>FMUL (scalar) — double-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0001</td>
<td>FDIV (scalar) — double-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0010</td>
<td>FADD (scalar) — double-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0011</td>
<td>FSUB (scalar) — double-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0100</td>
<td>FMAX (scalar) — double-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0101</td>
<td>FMIN (scalar) — double-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0110</td>
<td>FMAXNM (scalar) — double-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0111</td>
<td>FMINNM (scalar) — double-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>FMUL (scalar) — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1001</td>
<td>FADD (scalar) — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1010</td>
<td>FSUB (scalar) — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1011</td>
<td>FMAX (scalar) — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1100</td>
<td>FMIN (scalar) — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1101</td>
<td>FMAXNM (scalar) — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1110</td>
<td>FMINNM (scalar) — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>FMUL (scalar) — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>FCSEL — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>FCSEL — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0110</td>
<td>FCSEL — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1000</td>
<td>FCSEL — half-precision</td>
<td>-</td>
</tr>
</tbody>
</table>
## Floating-point data-processing (3 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>type</th>
<th>o1</th>
<th>o0</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>FMADD — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>FMSUB — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>FNMADD — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>FNMSUB — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>FMADD — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>FMSUB — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>FNMADD — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>FNMSUB — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>FMADD — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>FMSUB — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>FNMADD — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>FNMSUB — half-precision</td>
<td>ARMv8.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages.

Pseudocodes

Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```c
// AArch32.VCRMatch()
// ----------------

boolean AArch32.VCRMatch(bits(32) vaddress)

    if UsingAArch32() && ELUsingAArch32(EL1) && IsZero(vaddress<1:0>) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);
        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && !IsSecure() then
                match_word UInt(vaddress<4:2>) + 24 = '1';     // Non-secure vectors
            else
                match_word UInt(vaddress<4:2>) + 0 = '1';      // Secure vectors (or no EL3)
        else
            // Mask out bits not corresponding to vectors.
            if !HaveEL(EL3) then
                mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
            elsif !ELUsingAArch32(EL3) then
                mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
            else
                mask = '11011110':'00000000':'11011100':'11011110';
            match_word = match_word AND DBGVCR AND mask;
            match = !IsZero(match_word);    // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
            if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
                match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);
            else
                match = FALSE;
        return match;
```

Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```c
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// -------------------------------------------------------

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()

    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
```
// AArch32.BreakpointMatch()
// ----------------------------------
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
assert ELUsingAArch32(S1TranslationRegime());
assert n <= UInt(DBGDIDR.BRPs);

enabled = DBGBCR[n].E == '1';
ispriv = PSTATE.EL != ELO;
linked = DBGBCR[n].BT == '0x01';
isbreakpnt = TRUE;
linked_to = FALSE;

state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC, linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
(value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

if size == 4 then                  // Check second halfword
    // If the breakpoint address and BAS of an Address breakpoint match the address of the
    // second halfword of an instruction, but not the address of the first halfword, it is
    // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
    // event.
    (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
    if !value_match && match_i then
        value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
    if value_mismatch && !mismatch_i then
        value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
    // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
    // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
    // at the address DBGBVR[n]+2.
    if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
    if !value_mismatch then value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

match = value_match && state_match && enabled;
mismatch = value_mismatch && state_match && enabled;

return (match, mismatch);
AArch32.BreakpointValueMatch()

The first result is whether an Address Match or Context breakpoint is programmed on the
instruction at "address". The second result is whether an Address Mismatch breakpoint is
programmed on the instruction, that is, whether the instruction should be stepped.

(boolean, boolean) AArch32.BreakpointValueMatch(integer n, bits(32) vaddress, boolean linked_to)

"n" is the identity of the breakpoint unit to match against.
"vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
matching breakpoints.
"linked_to" is TRUE if this is a call from StateMatch for linking.

If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.

if n > UInt(DBGDIDR.BRPs) then
  (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIDR.BRPs), Unpredictable_BPNOTIMPL);
  assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
  if c == Constraint_DISABLED then return (FALSE, FALSE);

If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
call from StateMatch for linking).
if DBGBCR[n].E == '0' then return (FALSE, FALSE);

c = DBGBCR[n].C.

context_aware = (n >= UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));

If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
type = DBGBCR[n].BT;

if ((type IN {'011x', '11xx'}) && !HaveVirtHostExt()) ||       // Context matching
  (type == '010x' && !HaltOnBreakpointOrWatchpoint()) ||    // Address mismatch
  (type != '0x0x' && !context_aware) ||                     // Context matching
  (type == '1xxx' && !HaveEL(EL2))) then                    // EL2 extension
  (c, type) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
  assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
  if c == Constraint_DISABLED then return (FALSE, FALSE);

// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
mismatch_addr = (type == '0x0x');
mismatch_vm = (type == '011x');
mismatch_cid1 = (type == '10xx');
mismatch_cid2 = (type == '11xx');
linked = (type == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, of if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || mismatch_addr) then return (FALSE, FALSE);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !mismatch_addr then return (FALSE, FALSE);

// Do the comparison.
if mismatch_addr then
  byte = UInt(vaddress<1:0>);
  assert byte IN {0, 2}; // "vaddress" is halfword aligned
  byte_select_match = (DBGBCR[n].BAS<byte> == '1');
  BVR_match = vaddress<31:2> == DBGBVR[n]<31:2> && byte_select_match;
  elsif mismatch then
    BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
  elsif mismatch_cid1 then
    BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
  else
    Shared Pseudocode Functions
vmid = VTTBR_EL2.VMID;
bvr_vmid = DBGXVR[n]<15:0>;
BXVR_match = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} && vmid == bvr_vmid);
elsif match_cid2 then
  BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                 !ELUsingAArch32(EL2) &&
                 DBGXVR[n]<31:0> == CONTEXTIDR_EL2);
  bvr_match_valid = (match_addr || match_cid1);
  bxvr_match_valid = (match_vmid || match_cid2);
  match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);
return (match && !mismatch, !match && mismatch);
Library pseudocode for aarch32/debug/breakpoint/AArch32.StateMatch
// AArch32.StateMatch()
// ================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,  
  boolean isbreakpnt, boolean ispriv)
// "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

// If parameters are set to a reserved type, behaves as either disabled or a defined type
if ((HMC:SSC:PxC) IN {'011xx','100x0','101x0','11010','11101','1111x'}) ||       // Reserved
  (HMC == '0' && PxC == '00' && !isbreakpnt) ||                             // Usr/Svc/Sys
  (SSC IN {'01','10'} && !HaveEL(EL3)) || // No EL3
  (HMC:SSC:PxC == '11000' && !HaveEL(EL3)) ||                      // AArch64 only
  (HMC:SSC != '111' && SSC != '0' && PxC == '00' && !HaveEL(EL2)) ||  // No EL3/EL2
  (HMC:SSC:PxC == '11100' && !HaveEL(EL2))) then                            // No EL2
  (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);
assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
if c == Constraint_DISABLED then return FALSE;
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

PL2_match = HaveEL(EL2) && HMC == '1';
PL1_match = PxC<0> == '1';
PL0_match = PxC<1> == '1';
SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

el = PSTATE.EL;
if !ispriv && !isbreakpnt then
  priv_match = PL0_match;
elsif SSU_match then
  priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
else
  case el of
    when EL3 priv_match = PL1_match;    // EL3 and EL1 are both PL1
    when EL2 priv_match = PL2_match;
    when EL1 priv_match = PL1_match;
    when EL0 priv_match = PL0_match;
  case SSC of
    when '00' security_state_match = TRUE;   // Both
    when '01' security_state_match = !IsSecure(); // Non-secure only
    when '10' security_state_match = IsSecure(); // Secure only
    when '11' security_state_match = TRUE; // Both
  if linked then
    // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
    // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    lbn = UInt(LBN);
    first_ctx_cmp = UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
    last_ctx_cmp = UInt(DBGDIDR.BRPs);
    if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
      (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTX);
      assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
    case c of
      when Constraint_DISABLED return FALSE; // Disabled
      when Constraint_NONE linked = FALSE;   // No linking
      // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
      if linked then
        vaddress = bits(32) UNKNOWN;
        linked_to = TRUE;
        (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);
      return priv_match && security_state_match && (!linked || linked_match);
Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```java
// AArch32.GenerateDebugExceptions()
// ---------------------------------
boolean AArch32.GenerateDebugExceptions()
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());
```

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```java
// AArch32.GenerateDebugExceptionsFrom()
// -------------------------------------
boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)
    if from == EL0 && !ELStateUsingAArch32(EL1, secure) then
        mask = bit UNKNOWN; // PSTATE.D mask, unused for EL0 case
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);
    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    if HaveEL(EL3) && secure then
        spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
        if spd<1> == '1' then
            enabled = spd<0> == '1';
        else
            // SPD == 0b01 is reserved, but behaves the same as 0b00.
            enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from == EL0 then enabled = enabled || SDEI.SUIDEN == '1';
        enabled = from != EL2;
    return enabled;
```

Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```java
// AArch32.CheckForPMUOverflow()
// -----------------------------
// Signal Performance Monitors overflow IRQ and CTI overflow events
boolean AArch32.CheckForPMUOverflow()
    if !ELUsingAArch32(EL1) then return AArch64.CheckForPMUOverflow();
    pmuirq = PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSSET<31> == '1';
    for n = 0 to UInt(PMCR.N) - 1
        if HaveEL(EL2) then
            hpnn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPNN else HDCR.HPNN;
            hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
            E = (if n < UInt(hpnn) then PMCR.E else hpme);
        else
            E = PMCR.E;
        if E == '1' && PMINTENSET<n> == '1' && PMOVSSET<n> == '1' then pmuirq = TRUE;
    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);
    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)
    return pmuirq;
```
library pseudocode for aarch32/debug/pmu/AArch32.CountEvents

```java
// AArch32.CountEvents()
// ===========
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch32.CountEvents(integer n)
assert n == 31 || n < UInt(PMCR.N);

if (!ELUsingAAArch32(EL1) then return AArch64.CountEvents(n);
// Event counting is disabled in Debug state
debug = Halted();

// In Non-secure state, some counters are reserved for EL2
if HaveEL(EL2) then
    hpmn = if !ELUsingAAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
    hpme = if !ELUsingAAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
    E = if n < UInt(hpmn) || n == 31 then PMCR.E else hpme;
else
    E = PMCR.E;

enabled = E == '1' && PMCNTENSET<n> == '1';

if !IsSecure() then
    // Event counting in Non-secure state is allowed unless all of:
    // * EL2 and the HPMD Extension are implemented
    // * Executing at EL2
    // * PMNx is not reserved for EL2
    // * HDCR.HPMD == 1
    if HaveHPMDExt() && PSTATE.E == EL2 && (n < UInt(hpmn) || n == 31) then
        hpmd = if !ELUsingAAArch32(EL2) then MDCR_EL2.HPMD else HDCR.HPMD;
        prohibited = (hpmd == '1');
    else
        prohibited = FALSE;
else
    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    // * EL3 is using AArch32 and SDCR.SPME == 1
    // * Executing at EL0, and SDER.SUNIDEN == 1.
    spme = (if ELUsingAAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME);
    prohibited = HaveEL(EL3) && spme == '0' && (PSTATE.E != EL0 || SDER.SUNIDEN == '0');

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
if prohibited && !HaveNoSecurePMUDisableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // For the cycle counter, PMCR.DP enables counting when otherwise prohibited
if prohibited && n == 31 then prohibited = (PMCR.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
    filter = if n == 31 then PMCCFILTR else PMEVTYPER[n];

    P = filter<31>;
    U = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';

case PSTATE.E of
    when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
    when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
    when EL2 filtered = (NSH == '0');
    when EL3 filtered = (P == '1');

return !debug && enabled && !prohibited && !filtered;
```

Shared Pseudocode Functions
// AArch32.EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
SynchronizeContext();
assert HaveEL(EL2) & !IsSecure() & ELUsingAArch32(EL2);
AArch32.ReportHypEntry(exception);
AArch32.WriteMode(M32_Hyp);
SPSR[] = bits(32) UNKNOWN;
if HaveSSBSExt() then
PSTATE.SSBS = bits(1) UNKNOWN;
ELR_hyp = bits(32) UNKNOWN;
// In Debug state, the PE always execute T32 instructions when in AArch32 state, and
// PSTATE.(SS,A,I,F) are not observable so behave as UNKNOWN.
PSTATE.T = '1';                             // PSTATE.J is RES0
PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
PSTATE.E = HSCTRL.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
EDSCR.ERR = '1';
UpdateEDSCRFields();
EndOfInstruction();

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

// AArch32.EnterModeInDebugState()
// ===============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
SynchronizeContext();
assert ELUsingAArch32(EL1) & PSTATE.EL != EL2;
if PSTATE.M == M32_Monitor then SCR.NS = '0';
AArch32.WriteMode(target_mode);
SPSR[] = bits(32) UNKNOWN;
if HaveSSBSExt() then
PSTATE.SSBS = bits(1) UNKNOWN;
R[14] = bits(32) UNKNOWN;
// In Debug state, the PE always execute T32 instructions when in AArch32 state, and
// PSTATE.(SS,A,I,F) are not observable so behave as UNKNOWN.
PSTATE.T = '1';                             // PSTATE.J is RES0
PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
PSTATE.E = SCTLR.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
if HavePANExt() & SCTLR.SPAN == '0' then
PSTATE.PAN = '1';
EDSCR.ERR = '1';
UpdateEDSCRFields();                        // Update EDSCR processor state flags.
EndOfInstruction();
// AArch32.EnterMonitorModeInDebugState()
// -------------------------------------

// Take an exception in Debug state to Monitor mode.

// SynchronizeContext();
assert HaveEL(EL3) && ELUsingAArch32(EL3);
from_secure = IsSecure();
if PSTATE.M == M32_Monitor then SCR.NS = '0';
AArch32.WriteMode(M32_Monitor);
SPSR[] = bits(32) UNKNOWN;
if HaveSSBSExt() then PSTATE.SSBS = bits(1) UNKNOWN;
R[14] = bits(32) UNKNOWN;
// In Debug state, the PE always execute T32 instructions when in AArch32 state, and
// PSTATE.(SS,A,I,F) are not observable so behave as UNKNOWN.
PSTATE.T = '1'; // PSTATE.J is RES0
PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
PSTATE.E = SCTLR.EE;
PSTATE.I1 = '0';
PSTATE.IT = '00000000';
if HavePANExt() then
  if !from_secure then
    PSTATE.PAN = '0';
  elsif SCTLR.SPAN == '0' then
    PSTATE.PAN = '1';
  EDSCR.ERR = '1';
UpdateEDSCRFields(); // Update EDSCR processor state flags.
EndOfInstruction();
// AArch32.WatchpointByteMatch()
// --------------------------------------
boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)
bottom = if DBGWVR[n] &lt;= '1' then 2 else 3;  // Word or doubleword
byte_select_match = (DBGWCR[n].BAS&lt;UInt(vaddress&lt;bottom-1:0)&gt; &lt;'0');
mask = UInt(DBGWCR[n].MASK);

// If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '1111111', or
// DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
// UNPREDICTABLE.
if mask &gt; 0 &amp;&amp; !IsOnes(DBGWCR[n].BASE)
byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
else
  LSB = (DBGWCR[n].BASE AND NOT(DBGWCR[n].BASE - 1));  MSB = (DBGWCR[n].BASE + LSB);
  if !IsZero(MSB AND (MSB - 1)) then  // Not contiguous
    byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASECONTIGUOUS);
    bottom = 3;  // For the whole doubleword

// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask &gt; 0 &amp;&amp; mask &lt;= 2 then
  (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
  assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
case c of
  when Constraint_DISABLED return FALSE;  // Disabled
  when Constraint_NONE mask = 0;  // No masking
  // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value
  if mask &gt; bottom then
    WVR_match = (vaddress&lt;31:mask&gt; == DBGWVR[n]&lt;31:mask&gt;);
    // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
    if WVR_match &amp;&amp; !IsZero(DBGWVR[n]&lt;mask-1:bottom&gt;)
      WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
      WVR_match = vaddress&lt;31:bottom&gt; == DBGWVR[n]&lt;31:bottom&gt;;
  return WVR_match &amp;&amp; byte_select_match;

Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

// AArch32.WatchpointMatch()
// ------------------------
// Watchpoint matching in an AArch32 translation regime.
boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());
assert n &lt;= UInt(DBGDIDR.WRPs);

// "ispriv" is FALSE for LDRT/STRT instructions executed at EL1 and all
// load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
// loads.
enabled = DBGWCR[n].E == '1';
linked = DBGWCR[n].WT == '1';
isbreakpnt = FALSE;

state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
  linked, DBGWCR[n].LBN, isbreakpnt, ispriv);
ls_match = (DBGWCR[n].LSC&lt;if(iswrite then 1 else 0)&gt; == '1');

value_match = FALSE;
for byte = 0 to size - 1
  value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);
return value_match &amp;&amp; state_match &amp;&amp; ls_match &amp;&amp; enabled;

Library pseudocode for aarch32/debug/watchpoint/AAArch32.WatchpointMatch
Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
            (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);

Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception type, FaultRecord fault, bits(32) vaddress)

    exception = ExceptionSyndrome(type);
    d_side = type == Exception_DataAbort;

    exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = fault.ipaddress.NS;
        exception.ipaddress = ZeroExtend(fault.ipaddress.address);
    else
        exception.ipavalid = FALSE;
    return exception;

Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()

    bits(32) pc = ThisInstrAddr();
    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        // Generate an Alignment fault Prefetch Abort exception
        vaddress = pc;
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        secondstage = FALSE;
        AArch32.Abort(vaddress, AArch32.AlignmentFault(acctype, iswrite, secondstage));
AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

// The encoding used in the IFSR or DFSR can be Long-descriptor format or Short-descriptor
// format. Normally, the current translation table format determines the format. For an abort
// from Non-secure state to Monitor mode, the IFSR or DFSR uses the Long-descriptor format if
// any of the following applies:
// * The Secure TTBCR.EAE is set to 1.
// * The abort is synchronous and either:
//   - It is taken from Hyp mode.
//   - It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
long_format = FALSE;
if route_to_monitor && !IsSecure() then
    long_format = TTBCR_S.EAE == '1';
else
    long_format = TTBCR.EAE == '1';
d_side = TRUE;
if long_format then
    syndrome = AArch32.FaultStatusLD(d_side, fault);
else
    syndrome = AArch32.FaultStatusSD(d_side, fault);
if fault.acctype == AccType_IC then
    if (!long_format &&
        boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
        i_syndrome = syndrome;
        syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
    else
        i_syndrome = bits(32) UNKNOWN;
    if route_to_monitor then
        IFSR_S = i_syndrome;
    else
        IFSR = i_syndrome;
    if route_to_monitor then
        DFSR_S = syndrome;
    else
        DFSR = syndrome;
    DFAR_S = vaddress;
else
    DFSR = syndrome;
    DFAR = vaddress;
return;
Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```c
// AArch32.ReportPrefetchAbort()
// -------------------------------------------
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';
    d_side = FALSE;
    if long_format then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);
    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;
    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```c
// AArch32.TakeDataAbortException()
// -------------------------------------------

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                      (HCR.TGE == '1' || IsSecondStage(fault) ||
                      HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                      (IsDebugException(fault) && HDCR.TDE == '1'));
    preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;
    if IsDebugException(fault) then DBGDSRexxt.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Shared Pseudocode Functions
Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// ---------------------------------------

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
                    (SCR.TGE == '1' || IsSecondStage(fault)) ||
                    (HaveRASExt() && HCR.TEA == '1' && IsExternalAbort(fault)) ||
                    (IsDebugException(fault) && HCR.TDE == '1')));

bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0C;
lr_offset = 4;

if IsDebugException(fault) then DBGSCRext.MOE = fault.debugmoe;
if route_to_monitor then
    AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
    if fault.type == Fault_Alignment then             // PC Alignment fault
        exception = ExceptionSyndrome(Exception_PCAlignment);
        exception.vaddress = ThisInstrAddr();
    else
        exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
    end if
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    end if
else
    AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
    AArch32. EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/aborts/BranchTargetException

```
// BranchTargetException
// =====================
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
    route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception.BranchTarget);
    exception.syndrome<1:0> = PSTATE.BTYPE;
    exception.syndrome<24:2> = Zeros();        // RES0
    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Shared Pseudocode Functions
Page 1686
// EffectiveTCF()
// ==============
// Returns the TCF field applied to Tag Check Fails in the given Exception Level

bits(2) EffectiveTCF(bits(2) el)
if el == EL3 then
tcf = SCTLR_EL3.TCF;
elsif el == EL2 then
tcf = SCTLR_EL2.TCF;
elsif el == EL1 then
tcf = SCTLR_EL1.TCF;
elsif el == EL0 && HCR_EL2.<E2H,TGE> == '11' then
tcf = SCTLR_EL2.TCF0;
elsif el == EL0 && HCR_EL2.<E2H,TGE> != '11' then
tcf = SCTLR_EL1.TCF0;
return tcf;

// RecordTagCheckFail()
// =====================
// Records a tag fail exception into the appropriate TCFR_ELx

ReportTagCheckFail(bits(2) el, bit ttbr)
if el == EL3 then
  assert ttbr == '0';
  TFSR_EL3.TF0 = '1';
elsif el == EL2 then
  if ttbr == '0' then
    TFSR_EL2.TF0 = '1';
  else
    TFSR_EL2.TF1 = '1';
elsif el == EL1 then
  if ttbr == '0' then
    TFSR_EL1.TF0 = '1';
  else
    TFSR_EL1.TF1 = '1';
elsif el == EL0 then
  if ttbr == '0' then
    TFSRE0_EL1.TF0 = '1';
  else
    TFSRE0_EL1.TF1 = '1';

// TagCheckFail()
// ==============
// Handle a tag check fail condition

TagCheckFail(bits(64) vaddress, boolean iswrite)
  bits(2) tcf = EffectiveTCF(PSTATE.EL);
  if tcf == '01' then
    TagCheckFault(vaddress, iswrite);
  elsif tcf == '10' then
    ReportTagCheckFail(PSTATE.EL, vaddress<55>);
// TagCheckFault()
// ===============
// Raise a tag check fail exception.
TagCheckFault(bits(64) va, boolean write)
bits(2) target_el;
bits(64) preferred_exception_return = ThisInstrAddr();
integer vect_offset = 0x0;

if PSTATE.EL == EL0 then
target_el = if HCR_EL2.TGE == 0 then EL1 else EL2;
else
    target_el = PSTATE.EL;

exception = ExceptionSyndrome(Exception_DataAbort);
exception.syndrome<5:0> = '010001';
if write then
    exception.syndrome<6> = '1';
exception.vaddress = va;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

// AArch32.TakePhysicalFIQException()
// ----------------------------------
AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.FIQ == '1';

if route_to_aarch64 then AArch64.TakePhysicalFIQException();
route_to_hyp = (EL2Enabled() && PSTATE.EL IN (EL0, EL1) &&
    (HCR.TGE == '1' || HCR.FMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x1C;
lr_offset = 4;
if route_to_monitor then
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
    exception = ExceptionSyndrome(Exception_FIQ);
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
// AArch32.TakePhysicalIRQException()
// ---------------------------------
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';
    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN (EL0, EL1) && (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
// AArch32.TakePhysicalSErrorException()
// ----------------------------------------

AArch32.TakePhysicalSErrorException( boolean parity, bit extflag, bits(2) errortype,
                                      boolean impdef_syndrome, bits(24) full_syndrome)

ClearPendingPhysicalSError();
// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
  route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
  route_to_aarch64 = SCR_EL3.EA == '1';
if route_to_aarch64 then
  AArch64.TakePhysicalSErrorException(impdef_syndrome, full_syndrome);
route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0,EL1} && (HCR.TGE == '1' || HCR.AMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x10;
lr_offset = 8;

fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
vaddress = bits(32) UNKNOWN;
if route_to_monitor then
  AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
  AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
  exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
  if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
  else
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
  end else
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
  AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException

// AArch32.TakeVirtualFIQException()
// ---------------------------------}

AArch32.TakeVirtualFIQException()  
assert EL2Enabled() && PSTATE.EL IN {EL0,EL1};
if ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and FMO==1
  assert HCR.TGE == '0' && HCR.FMO == '1';
else
  assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
// Check if routed to AArch64 state
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x1C;
lr_offset = 4;
AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
// AArch32.TakeVirtualIRQException()
// ---------------------------------

AArch32.TakeVirtualIRQException()
assert EL2Enabled() && PSTATE.EL IN {EL0, EL1};
if ELUsingAArch32(EL2) then /* Virtual IRQs enabled if TGE==0 and IMO==1 */
    assert HCR.TGE == '0' && HCR.IMO == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';
// Check if routed to AArch64 state
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
    AArch64.TakeVirtualIRQException();

bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x18;
lr_offset = 4;
AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualSErrorException

// AArch32.TakeVirtualSErrorException(bit extflag, bits(2) errortype, boolean impdef_syndrome, bits(24) full_syndrome)
// -----------------------------------------------

AArch32.TakeVirtualSErrorException(bit extflag, bits(2) errortype, boolean impdef_syndrome, bits(24) full_syndrome)
assert EL2Enabled() && PSTATE.EL IN {EL0, EL1};
if ELUsingAArch32(EL2) then /* Virtual SError enabled if TGE==0 and AMO==1 */
    assert HCR.TGE == '0' && HCR.AMO == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
// Check if routed to AArch64 state
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
    AArch64.TakeVirtualSErrorException(impdef_syndrome, full_syndrome);
route_to_monitor = FALSE;
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x18;
lr_offset = 8;
vaddress = bits(32) UNKNOWN;
parity = FALSE;
if HaveRASExt() then
    if ELUsingAArch32(EL2) then
        fault = AArch32.AsynchExternalAbort(FALSE, VDSR.AET, VDSR.ExT);
    else
        fault = AArch32.AsynchExternalAbort(FALSE, VSES_EL2.AET, VSES_EL2.ExT);
    else
        fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
ClearPendingVirtualSErrorException();
AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
AArch32.SoftwareBreakpoint()  // Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

AArch32.SoftwareBreakpoint(bits(16) immediate)

if (EL2Enabled() && !ELUsingAArch32(EL2) &&
    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
    AArch64.SoftwareBreakpoint(Immediate);

vaddress = bits(32) UNKNOWN;
acctype = AccType_IFETCH;           // Take as a Prefetch Abort
iswrite = FALSE;
entry = DebugException_BKPT;
fault = AArch32.DebugFault(acctype, iswrite, entry);
AArch32.Abort(vaddress, fault);

Library pseudocode for aarch32/exceptions/debug/DebugException

constant bits(4) DebugException_Breakpoint  = '0001';
constant bits(4) DebugException_BKPT        = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint  = '1010';

Library pseudocode for aarch32/exceptions/exceptions/AArch32.CheckAdvSIMDOrFPRegisterTraps

AArch32.CheckAdvSIMDOrFPRegisterTraps()  // Library pseudocode for aarch32/exceptions/exceptions/AArch32.CheckAdvSIMDOrFPRegisterTraps()

// Check if an instruction that accesses an Advanced SIMD and floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

if PSTATE.EL == EL1 && EL2Enabled() then
    tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
    tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

    if (tid0 == '1' && reg == '0000')                             // FPSID
        || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then    // MVFRx
            if ELUsingAArch32(EL2) then
                AArch32.AArch32SystemAccessTrap(EL2, ThisInstr());
            else
                AArch64.AArch32SystemAccessTrap(EL2, ThisInstr());

Shared Pseudocode Functions
// AArch32.ExceptionClass()
// Return the Exception Class and Instruction Length fields for reported in HSR

(int, bit) AArch32.ExceptionClass(Exception type)

    il = if ThisInstrLength() == 32 then '1' else '0';

    case type of
        when Exception_Uncategorized ec = 0x00; il = '1';
        when Exception_WFxTrap ec = 0x01;
        when Exception_CP15RRTTrap ec = 0x03;
        when Exception_CP15RRTTrap ec = 0x04;
        when Exception_CP14RTTrap ec = 0x05;
        when Exception_CP14RTTrap ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap ec = 0x07;
        when Exception_FPIDTrap ec = 0x08;
        when Exception_PACTrap ec = 0x09;
        when Exception_CP14RRTrap ec = 0x0C;
        when Exception_BranchTarget ec = 0x0D;
        when Exception_IllegalState ec = 0x0E; il = '1';
        when Exception_SupervisorCall ec = 0x11;
        when Exception_HypervisorCall ec = 0x12;
        when Exception_MonitorCall ec = 0x13;
        when Exception_ERetTrap ec = 0x1A;
        when Exception_InstructionAbort ec = 0x20; il = '1';
        when Exception_DataAbort ec = 0x24;
        when Exception_NV2DataAbort ec = 0x25;
        when Exception_FPTrappedException ec = 0x28;
        otherwise Unreachable();

        if ec IN {0x20, 0x24} && PSTATE.EL == EL2 then
            ec = ec + 1;

        return (ec, il);

// AArch32.GeneralExceptionsToAArch64()
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2 is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()

    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
                (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));

Shared Pseudocode Functions
// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.
AArch32.ReportHypEntry(ExceptionRecord exception)
    Exception type = exception.type;
    (ec,il) = AArch32.ExceptionClass(type);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';
    HSR = ec<5:0>:il:iss;

    if type IN {Exception InstructionAbort, Exception PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif type == Exception DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

        if exception.ipavalid then
            HPFAR<31:4> = exception.ipaddress<39:12>;
        else
            HPFAR<31:4> = bits(28) UNKNOWN;
    return;

// AArch32.ResetControlRegisters(boolean cold_reset);

Shared Pseudocode Functions
AArch32.TakeReset(boolean cold_reset)
    assert HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0'; // Secure state
    elsif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset the CP14 and CP15 registers and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111'; // All asynchronous exceptions masked
    PSTATE.IT = '00000000'; // IT block state reset
    PSTATE.T = SCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
    PSTATE.E = SCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv; // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
        if MVBAR<0> == '1' then // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';
    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');
    BranchTo(rv, BranchType_RESET);

Libary pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFF0000
        return Ones(16) : Zeros(16);
    else
        return VBAR<31:5> : Zeros(5);
Library pseudocode for aarch32/exceptions/ieeefp/AArch32.FPTrappedException

// AArch32.FPTrappedException()
// -----------------------------------------
AArch32.FPTrappedException(bits(8) accumulated_exceptions)
  if AArch32.GeneralExceptionsToAArch64() then
    is_ase = FALSE;
    element = 0;
    AArch64.FPTrappedException(is_ase, element, accumulated_exceptions);
  FPESC.DEX    = '1';
  FPESC.TFV    = '1';
  FPESC<7:4>   = accumulated_exceptions<7:4>;                  // IDF,IXF,UFF,OFF,DZF,IOF
  FPESC<10:8>  = '111';                                          // VECITR is RES1
  AArch32.TakeUndefInstrException();

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

// AArch32.CallHypervisor()
// --------------------------
// Performs a HVC call
AArch32.CallHypervisor(bits(16) immediate)
  assert HaveEL(EL2);
  if !ELUsingAArch32(EL2) then
    AArch64.CallHypervisor(immediate);
  else
    AArch32.TakeHVCException(immediate);

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

// AArch32.CallSupervisor()
// -------------------------
// Calls the Supervisor
AArch32.CallSupervisor(bits(16) immediate)
  if AArch32.CurrentCond() != '1110' then
    immediate = bits(16) UNKNOWN;
  if AArch32.GeneralExceptionsToAArch64() then
    AArch64.CallSupervisor(immediate);
  else
    AArch32.TakeSVCException(immediate);

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCException

// AArch32.TakeHVCException()
// ---------------------------
AArch32.TakeHVCException(bits(16) immediate)
  assert HaveEL(EL2) \&\& ELUsingAArch32(EL2);
  AArch32.ITAdvance();
  SSAdvance();
  bits(32) preferred_exception_return = NextInstrAddr();
  vect_offset = 0x08;
  exception = ExceptionSyndrome(Exception_HypervisorCall);
  exception.syndrome<15:0> = immediate;
  if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
  else
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);

Shared Pseudocode Functions
Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCException

// AArch32.TakeSMCException()
// =========================

AArch32.TakeSMCException()
assert HaveEL(EL3) && ELUsingAArch32(EL3);
AArch32.ITAdvance();
SSAdvance();
bits(32) preferred_exception_return = NextInstrAddr();
vect_offset = 0x08;
lr_offset = 0;
AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCException

// AArch32.TakeSVCException()
// ==========================

AArch32.TakeSVCException(bits(16) immediate)
AArch32.ITAdvance();
SSAdvance();
route_to_hyp = EL2Enabled() && PSTATE.EL == EL0 && HCR.TGE == '1';
bits(32) preferred_exception_return = NextInstrAddr();
vect_offset = 0x08;
lr_offset = 0;
if PSTATE.EL == EL2 || route_to_hyp then
    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
    AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
// AArch32.EnterHypMode()
// ---------------------
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return, integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    spsr = GetPSRFromPSTATE();
    if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
        AArch32.WriteMode(M32_Hyp);
        SPSR[] = spsr;
    if HaveSSBSExt() then PSTATE.SSBS = HSCTLR.DSSBS;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE;                       // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.I = '0';
    if target_mode IN {M32_FIQ} then
        PSTATE.<A,I>F = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.I = '0';
    PSTATE.IT = '00000000';
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);
EndOfInstruction();

// AArch32.EnterMode()
// -------------------
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset, integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.I = '0';
    if HavePANExt() && SCTLR.SPAN == '0' then
        PSTATE.PAN = '1';
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);
EndOfInstruction();
// AArch32.EnterMonitorMode()
// ----------------------------------------------
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset, integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) & ELUsingAArch32(EL3);
    from_secure = IsSecure();
    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    if HaveSSBSExt() then PSTATE.SBS = SCTLR.DSSBS;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
        BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);
    EndOfInstruction();

Library pseudocode for aarch32/exceptions/traps/AArch32.AArch32SystemAccessTrap

// AArch32.AArch32SystemAccessTrap()
// ----------------------------------------------
// Trapped AArch32 System register access other than due to CPTR_EL2 or CPACR_EL1.

AArch32.AArch32SystemAccessTrap(bits(2) target_el, bits(32) instr)
    assert HaveEL(target_el) & target_el != EL0 & Uint(target_el) >= Uint(PSTATE.EL);
    if !ELUsing(target_el) || AArch32.GeneralExceptionsToAArch64() then
        AArch64.AArch32SystemAccessTrap(target_el, instr);
    assert target_el IN {EL1, EL2};
    if target_el == EL2 then
        exception = AArch32.AArch32SystemAccessTrapSyndrome(instr);
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
ExceptionRecord AArch32.AArch32SystemAccessTrapSyndrome(bits(32) instr)

ExceptionRecord exception;
cpnum = UInt(instr<11:8>);

bits(20) iss = Zeros();
if instr<27:24> == '1110' && instr<4> == '1' && instr<31:28> != '1111' then
  // MRC/MCR
  case cpnum of
    when 10 exception = ExceptionSyndrome(Exception_FPIDTrap);
    when 14 exception = ExceptionSyndrome(Exception_CP14RTTrap);
    when 15 exception = ExceptionSyndrome(Exception_CP15RTTrap);
  otherwise Unreachable();
  iss<19:17> = instr<7:5>; // opc2
  iss<16:14> = instr<23:21>; // opc1
  iss<13:10> = instr<19:16>; // CRn
  iss<8:5> = instr<15:12>; // Rt
  iss<4:1> = instr<3:0>; // CRm
elsif instr<27:21> == '110010' && instr<31:28> != '1111' then
  // MRRC/MCRR
  case cpnum of
    when 14 exception = ExceptionSyndrome(Exception_CP14RRTTrap);
    when 15 exception = ExceptionSyndrome(Exception_CP15RRTTrap);
  otherwise Unreachable();
  iss<19:16> = instr<7:4>; // opc1
  iss<13:10> = instr<19:16>; // Rt2
  iss<8:5> = instr<15:12>; // Rt
  iss<4:1> = instr<3:0>; // CRm
elsif instr<27:25> == '110' && instr<31:28> != '1111' then
  // LDC/STC
  assert cpnum == 14;
  exception = ExceptionSyndrome(Exception_CP14DTTrap);
  iss<19:12> = instr<7:0>; // imm8
  iss<4> = instr<23>; // U
  iss<21> = instr<24,21>; // P,W
if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
  iss<8:5> = bits(4) UNKNOWN;
  iss<3> = '1';
else
  iss<8:5> = instr<19:16>; // Rn
  iss<3> = '0';
else
  Unreachable();
iss<0> = instr<20>; // Direction

exception.syndrome<24:20> = ConditionSyndrome();
exception.syndrome<19:0> = iss;

return exception;
// AArch32.CheckAdvSIMDOrFPEnabled()
// ----------------------------------------
// Check against CPACR, FPEXC, HCPR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)
    if PSTATE.EL == EL0 && (!HaveEL(EL2) || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0') && !ELUsingAArch32(EL1)) then
        // The PE behaves as if FPEXC.EN is 1
        AArch64.CheckFPAdvSIMDEnabled();
    elsif PSTATE.EL == EL0 && HaveEL(EL2) && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && !ELUsingAArch32(EL1) then
        if fpexc_check && HCR_EL2.RW == '0' then
            fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
        if fpexc_en == '0' then UNDEFINED;
        AArch64.CheckFPAdvSIMDEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL(EL3) && ELUsingAAArch32(EL3) && !IsSecure() then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;

            if cpacr_cp10 == '10' then
                (c, cpacr_cp10) = ConstrainUnpredictableBits(Unpredictable_RESCPACR);

        // Check if access disabled in CPACR
        case cpacr_cp10 of
            when '00' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
            if disabled then UNDEFINED;

        // If required, check FPEXC enabled bit.
        if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPR and CPTR_EL3
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// -------------------------------
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if HaveEL(EL2) && !IsSecure() then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;
            if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';
            // Check if access disabled in HCPT
            if advsimd && hcptr_tase == '1' || hcptr_cp10 == '1' then
                exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                exception.syndrome<24:20> = ConditionSyndrome();
                if advsimd then
                    exception.syndrome<5> = '1';
                else
                    exception.syndrome<5> = '0';
                end
                exception.syndrome<3:0> = '1010';
                if PSTATE.EL == EL2 then
                    AArch32.TakeUndefInstrException(exception);
                else
                    AArch32.TakeHypTrapException(exception);
                end
            end
        end
if HaveEL(EL3) && !ELUsingAArch32(EL3) then
    // Check if access disabled in CPTR_EL3
    if CPTR_EL3.TFP == '1' then
        AArch64.AdvSIMDFPAccessTrap(EL3);
    end
return;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

```
// AArch32.CheckForSMCUndefOrTrap()
// -------------------------------
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;
    end
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUndefOrTrap(Zeros(16));
    else
        route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            exception = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(exception);
        end
    end
return;
```
// AArch32.CheckForWFxTrap()  
// --------------------------
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
assert HaveEL(target_el);

// Check for routing to AArch64
if !ELUsingAArch32(target_el) then
    AArch64.CheckForWFxTrap(target_el, is_wfe);
    return;
endcase

if target_el == EL1
    trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
elsif target_el == EL2
    trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
elsif target_el == EL3
    trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

if trap then
    if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.WFxTrap(target_el, is_wfe);
    elsif target_el == EL2
        exception = ExceptionSyndrome(Exception_WFxTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        exception.syndrome<0> = if is_wfe then '1' else '0';
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
    endcase
endif

// AArch32.CheckITEnabled()  
// --------------------------
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
if PSTATE.EL == EL2 then
    it_disabled = HSCTLR.ITD;
else
    it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[].ITD);
endif
if it_disabled == '1' then
    if mask != '1000' then UNDEFINED;
    // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
    next_instr = AArch32.MemSingle(NextInstrAddr(), 2, AccType_IFETCH, TRUE);
    if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx', '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
        // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
        // taken on the IT instruction or the next instruction. This is not reflected in
        // the pseudocode, which always takes the exception on the IT instruction. This
        // also does not take into account cases where the next instruction is UNPREDICTABLE.
        UNDEFINED;
    endif
endif
return;

// Shared Pseudocode Functions
Page 1703
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

// AArch32.CheckIllegalState()
// ----------------------------------
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = EL2Enabled() && PSTATE.EL == EL0 && HCR.TGE == '1';
        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;
        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
            else
                AArch32.TakeUndefInstrException();

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

// AArch32.CheckSETENDEnabled()
// --------------------------
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[].SED);
        if setend_disabled == '1' then
            UNDEFINED;
            return;

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

// AArch32.TakeHypTrapException()
// -----------------------------
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x14;
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

// AArch32.TakeMonitorTrapException()
// --------------------------------
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```cpp
// AArch32.TakeUndefInstrException()
// ---------------------------------
AArch32.TakeUndefInstrException()
exception = ExceptionSyndrome(Exception_Uncategorized);
AArch32.TakeUndefInstrException(exception);

AArch32.TakeUndefInstrException(ExceptionRecord exception)
route_to_hyp = EL2Enabled() && PSTATE.EL == EL0 && HCR.TGE == '1';
bits(32) preferred_exception_return = ThisInstrAddr();
vec_offset = 0x04;
lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vec_offset);
elsif route_to_hyp then
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
    AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vec_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

```cpp
// AArch32.UndefinedFault()
// ========================
AArch32.UndefinedFault()
if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();
AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/functions/aborts/AArch32.CreateFaultRecord

```cpp
// AArch32.CreateFaultRecord()
// ---------------------------
FaultRecord AArch32.CreateFaultRecord(Fault type, bits(40) ipaddress, bits(4) domain,
    integer level, AccType acctype, boolean write, bit extflag,
    bits(4) debugmoe, bits(2) errortype, boolean secondstage, boolean s2fs1walk)
    FaultRecord fault;
fault.type = type;
if (type != Fault_None && PSTATE.EL != EL2 && TTBCR.EAE == '0' && !secondstage && !s2fs1walk &&
    AArch32.DomainValid(type, level)) then
    fault.domain = domain;
else
    fault.domain = bits(4) UNKNOWN;
fault.debugmoe = debugmoe;
fault.errortype = errortype;
fault.ipaddress.address = ZeroExtend(ipaddress);
fault.level = level;
fault.acctype = acctype;
fault.write = write;
fault.extflag = extflag;
fault.secondstage = secondstage;
fault.s2fs1walk = s2fs1walk;
return fault;
```
// AArch32.DomainValid()
// -------------------------
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.
boolean AArch32.DomainValid(Fault type, integer level)
    assert type != Fault_None;
    case type of
        when Fault_Domain
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk,
            Fault_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;

// AArch32.FaultStatusLD()
// -------------------------
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;
    bits(32) fsr = Zeros();
    if HaveRASExt() & IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then fsr<12> = fault.extflag;
        fsr<9> = '1';
        fsr<5:0> = EncodeLDFSC(fault.type, fault.level);
    return fsr;

// AArch32.FaultStatusSD()
// -------------------------
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;
    bits(32) fsr = Zeros();
    if HaveRASExt() & IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then fsr<12> = fault.extflag;
        fsr<9> = '0';
        fsr<10,3:0> = EncodeSDFSC(fault.type, fault.level);
        if d_side then
            fsr<7:4> = fault.domain; // Domain field (data fault only)
    return fsr;
// AArch32.FaultSyndrome()
// --------------------------------
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
assert fault.type != Fault_None;

bits(25) iss = Zeros();
if HaveRASExt() && IsAsyncAbort(fault) then iss<11:10> = fault.errortype; // AET
if d_side then
  if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
  if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC, AccType_AT} then
    iss<8> = '1';  iss<6> = '1';
  else
    iss<6> = if fault.write then '1' else '0';
  if IsExternalAbort(fault) then iss<9> = fault.extflag;
  iss<7> = if fault.s2fs1walk then '1' else '0';
  iss<5:0> = EncodeLDFSC(fault.type, fault.level);
return iss;

Shared Pseudocode Functions
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

default EncodeSDFSC(Fault type, integer level)
bits(5) result;
case type of
  when Fault_AccessFlag
    assert level IN {1,2};
    result = if level == 1 then '00011' else '00110';
  when Fault_Alignment
    result = '00001';
  when Fault_Permission
    assert level IN {1,2};
    result = if level == 1 then '01101' else '01111';
  when Fault_Domain
    assert level IN {1,2};
    result = if level == 1 then '01001' else '01011';
  when Fault_Translation
    assert level IN {1,2};
    result = if level == 1 then '00101' else '00111';
  when Fault_SyncExternal
    result = '01000';
  when Fault_SyncExternalOnWalk
    assert level IN {1,2};
    result = if level == 1 then '01100' else '01110';
  when Fault_SyncParity
    result = '11001';
  when Fault_SyncParityOnWalk
    assert level IN {1,2};
    result = if level == 1 then '11100' else '11110';
  when Fault_AsyncParity
    result = '11000';
  when Fault_AsyncExternal
    result = '10110';
  when Fault_Debug
    result = '00010';
  when Fault_TLBConflict
    result = '10000';
  when Fault_Lockdown
    result = '10100'; // IMPLEMENTATION DEFINED
  when Fault_Exclusive
    result = '10101'; // IMPLEMENTATION DEFINED
  when Fault_ICacheMaint
    result = '00100';
otherwise
  Unreachable();

return result;

Library pseudocode for aarch32/functions/common/A32ExpandImm

// A32ExpandImm()
// ==============

void A32ExpandImm(bits(12) imm12)

  // PSTATE.C argument to following function call does not affect the imm32 result.
  (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

return imm32;
Library pseudocode for aarch32/functions/common/A32ExpandImm_C

// A32ExpandImm_C()
// ================
(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

  unrotated_value = ZeroExtend(imm12<7:0>, 32);
  (imm32, carry_out) = Shift_C(unrotated_value, SRTYPE_ROR, 2*UInt(imm12<11:8>), carry_in);
  return (imm32, carry_out);

Library pseudocode for aarch32/functions/common/DecodeImmShift

// DecodeImmShift()
// ================
(SRTYPE, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

case type of
  when '00'
    shift_t = SRTYPE_LSL;  shift_n = UInt(imm5);
  when '01'
    shift_t = SRTYPE_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
  when '10'
    shift_t = SRTYPE_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
  when '11'
    if imm5 == '00000' then
      shift_t = SRTYPE_RRX;  shift_n = 1;
    else
      shift_t = SRTYPE_ROR;  shift_n = UInt(imm5);
  return (shift_t, shift_n);

Library pseudocode for aarch32/functions/common/DecodeRegShift

// DecodeRegShift()
// ================
SRTYPE DecodeRegShift(bits(2) type)

case type of
  when '00'  shift_t = SRTYPE_LSL;
  when '01'  shift_t = SRTYPE_LSR;
  when '10'  shift_t = SRTYPE_ASR;
  when '11'  shift_t = SRTYPE_ROR;
return shift_t;

Library pseudocode for aarch32/functions/common/RRX

// RRX()
// -----

bits(N) RRX(bits(N) x, bit carry_in)
  (result, -) = RRX_C(x, carry_in);
  return result;

Library pseudocode for aarch32/functions/common/RRX_C

// RRX_C()
// -------

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
  result = carry_in : x<N-1:1>;
  carry_out = x<0>;
  return (result, carry_out);
Library pseudocode for aarch32/functions/common/SRType

```plaintext
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

Library pseudocode for aarch32/functions/common/Shift

```plaintext
// Shift()
// ========

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
  (result, -) = Shift_C(value, type, amount, carry_in);
return result;
```

Library pseudocode for aarch32/functions/common/Shift_C

```plaintext
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
assert !(type == SRType_RRX && amount != 1);
if amount == 0 then
  (result, carry_out) = (value, carry_in);
else
  case type of
  when SRType_LSL
    (result, carry_out) = LSL_C(value, amount);
  when SRType_LSR
    (result, carry_out) = LSR_C(value, amount);
  when SRType_ASR
    (result, carry_out) = ASR_C(value, amount);
  when SRType_ROR
    (result, carry_out) = ROR_C(value, amount);
  when SRType_RRX
    (result, carry_out) = RRX_C(value, carry_in);
  return (result, carry_out);
```
// Library pseudocode for aarch32/functions/common/T32ExpandImm_C

// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

if imm12<11:10> == '00' then
    case imm12<9:8> of
        when '00'
            imm32 = ZeroExtend(imm12<7:0>, 32);
        when '01'
            imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
        when '10'
            imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
        when '11'
            imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
    carry_out = carry_in;
else
    unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
    (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

return (imm32, carry_out);

// Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

// AArch32.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained CP15 traps in HSTR and HCR.

boolean AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if EL2Enabled() && PSTATE.EL IN {EL0,EL1} then
        if PSTATE.EL == EL0 && !ELUsingAArch32(EL2) then
            return AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !(major IN {4,14}) && HSTR<major> == '1' then
            return TRUE;
        // Check for MRC and MCR disabled by HCR.TIDCP
        if (HCR.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9 && CRm IN {0,1,2,5,6,7,8}) ||
             (CRn == 10 && CRm IN {0,1,4,8}) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
Library pseudocode for aarch32/functions/coproc/AArch32.CheckSystemAccess
// AArch32.CheckSystemAccess()
// ----------------------------------
// Check System register access instruction for enables and disables

AArch32.CheckSystemAccess(integer cp_num, bits(32) instr)
assert cp_num == UInt(instr<11:8>) && (cp_num IN {14,15});
if PSTATE.EL == EL0 && !ELUsingAArch32 (EL1) then
  AArch64.CheckAArch32SystemAccess(instr);
  return;
// Decode the AArch32 System register access instruction
if instr<31:28> !='1111' && instr<27:24> == '1110' && instr<4> == '1' then      // MRC/MCR
  cprrt = TRUE;  cpdt = FALSE;  nreg = 1;
  opc1 = UInt(instr<23:21>);
  opc2 = UInt(instr<7:5>);
  CRn = UInt(instr<19:16>);
  CRm = UInt(instr<3:0>);
elsif instr<31:28> != '1111' && instr<27:21> == '1100010' then                   // MRRC/MCRR
  cprrt = TRUE;  cpdt = FALSE;  nreg = 2;
  opc1 = UInt(instr<7:4>);
  CRm = UInt(instr<3:0>);
elsif instr<31:28> != '1111' && instr<27:25> == '110' && instr<22> == '0' then   // LDC/STC
  cprrt = FALSE;  cpdt = TRUE;  nreg = 0;
  opc1 = 0;
  CRn = UInt(instr<15:12>);
else
  allocated = FALSE;
//
// Coarse-grain decode into CP14 or CP15 encoding space. Each of the CPxxxInstrDecode functions
// returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
if cp_num == 14 then
  // LDC and STC only supported for c5 in CP14 encoding space
  if cpdt && CRn != 5 then
    allocated = FALSE;
  else
    // Coarse-grained decode of CP14 based on opc1 field
    case opc1 of
      when 0     allocated = CP14DebugInstrDecode(instr);
      when 1     allocated = CP14TraceInstrDecode(instr);
      when 7     allocated = CP14JazelleInstrDecode(instr);    // JIDR only
    otherwise allocated = FALSE;          // All other values are unallocated
  end case;
elsif cp_num == 15 then
  // LDC and STC not supported in CP15 encoding space
  if !cprrt then
    allocated = FALSE;
  else
    allocated = CP15InstrDecode(instr);
// Coarse-grain traps to EL2 have a higher priority than exceptions generated because
// the access instruction is UNDEFINED
if AArch32.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
  // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
  // User mode is UNDEFINED when the trap is disabled, then it is
  // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
  // when the trap is enabled.
  if PSTATE.EL == EL0 && EL2Enabled() && !allocated then
    if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0" then
      UNDEFINED;
      AArch32.AArch32SystemAccessTrap(EL2, instr);
    else
      allocated = FALSE;
    end if;
  end if;
else
  allocated = FALSE;
if !allocated then
  UNDEFINED;
// If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
AArch32.CheckSystemAccessTraps(instr);
return;
AArch32.CheckSystemAccessEL1Traps(bits(32) instr)
assert PSTATE.EL == EL0;
if ((HaveEL(EL1) && IsSecure() && ELUsingAArch32(EL1)) || IsInHost()) then
    AArch64.CheckAArch32SystemAccessEL1Traps(instr);
return;
trap = FALSE;
// Decode the AArch32 System register access instruction
[op, cp_num, opc1, CRn, CRm, opc2, write] = AArch32.DecodeSysRegAccess(instr);
if cp_num == 14 then
    if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 == 0) || // DBGDTRRXint/DBGDTRTXint
        (op == SystemAccessType_RT && opc1 == 0 && CRn == 5 && opc2 == 0)) then
        trap = !Halted() && DBGDSCRext.UDCCdis == '1';
    elsif opc1 == 0 then
        trap = DBGDSCRext.UDCCdis == '1';
    elsif opc1 == 1 then
        trap = CPACR.TRCDIS == '1';
        if HaveEL(EL3) && ELUsingAArch32(EL3) && NSACR.NSTRCDIS == '1' then
            trap = TRUE;
        elsif cp_num == 15 then
            if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0) || // PMCR
                (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 1) || // PMCNTENSET
                (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 2) || // PMCNTENCLR
                (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 3) || // PMOVSR
                (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 6) || // PMCEID0
                (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 7) || // PMCEID1
                (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 12)) then          // PMEVTYPER<n>
                trap = PMUSERENR.EN == '0';
            elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 14 && opc2 == 4 then // PMSWINC
                trap = PMUSERENR.EN == '0' && PMUSERENR.SW == '0';
            elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 0) || // PMCCNTR
                (op == SystemAccessType_RRT && opc1 == 0 && CRm == 9)) then                          // PMCCNTR (MRRC/MCRR)
                trap = PMUSERENR.EN == '0' && (write || PMUSERENR.CR == '0');
            elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 2) || // PMXEVCNTR
                (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8 && CRm <= 11)) then  // PMEVCNTR<n>
                trap = PMUSERENR.EN == '0' && (write || PMUSERENR.ER == '0');
            elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 5 then    // PMSELR
                trap = PMUSERENR.EN == '0' && PMUSERENR.ER == '0';
            elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then // CNTP_TVAL CNTP_CTL CNTP_CVAL
                trap = CNTKCTL.PL0PTEN == '0';
            elsif op == SystemAccessType_RRT && opc1 == 1 && CRm == 14 then                               // CNTVCT
                trap = CNTKCTL.PL0VCTEN == '0';
        else
            trap = PMUSERENR.EN == '0' && PMUSERENR.SW == '0';
        end
    else
        trap = PMUSERENR.EN == '0' && PMUSERENR.CR == '0';
    end
else
    trap = PMUSERENR.EN == '0';
end
if trap then
    AArch32.AArch32SystemAccessTrap(EL1, instr);
// AArch32.CheckSystemAccessEL2Traps()
// Check for configurable traps to EL2 of a System register access instruction.

AArch32.CheckSystemAccessEL2Traps(bits(32) instr)
assert EL2Enabled() && PSTATE.EL IN {EL0, EL1, EL2};

if EL2Enabled() && !ELUsingAArch32(EL2) then
  AArch64.CheckAArch32SystemAccessEL2Traps(instr);
  return;

trap = FALSE;

// Decode the AArch32 System register access instruction
(op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

if cp_num == 14 && PSTATE.EL IN {EL0, EL1} then
  if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 0) || // DBGDRAR
      (op == SystemAccessType_RRT && opc1 == 0 && CRm == 2)) then
    trap = HDCR.TDRA == '1' || HDCR.TDE == '1' || HCR.TGE == '1';
  elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 4) || // DBGOSLAR
      (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 4) ||    // DBGOSLSR
      (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 4) ||    // DBGOSDLR
      (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 4 && opc2 == 4)) then // DBGPRCR
    trap = HDCR.TDOSA == '1' || HDCR.TDE == '1' || HCR.TGE == '1';
  elsif opc1 == 0 then
    if Halted() || !(op == SystemAccessType_RT && CRn == 0 && CRm == 5 && opc2 == 0) then
      trap = HDCR.TDA == '1' || HDCR.TDE == '1' || HCR.TGE == '1';
    elsif opc1 == 1 then
      trap = HCPTR.TTA == '1';
    elseif HaveEL(EL3) && ELUsingAArch32(EL3) && NSACR.NSTRCDIS == '1' then
      trap = TRUE;
    elsif op == SystemAccessType_RT && opc1 == 7 && CRn == 0 && CRm == 0 && opc2 == 0 then  // JIDR
      trap = HCR.TID0 == '1';
  else
    if cp_num == 15 && PSTATE.EL IN {EL0, EL1} then
      if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 0) ||  // SCTLR
          (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 0) ||  // TTBR0
          (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 2)) then
        trap = HCR.TID0 == '1';
      elsif op == SystemAccessType_RT && opc1 == 7 && CRn == 0 && CRm == 0 && opc2 == 0 then // JIDR
        trap = HCR.TID0 == '1';
    else
      if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 0) ||  // TTBR0
          (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 2) ||            // TTBR0
          (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 2)) then
        trap = HCR.TID0 == '1';
      elseif op == SystemAccessType_RT && opc1 == 7 && CRn == 0 && CRm == 0 && opc2 == 0 then // JIDR
        trap = HCR.TID0 == '1';
      endif
    endif
  endif
endif

Shared Pseudocode Functions
elsif (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 6 && opc2 == 1) ||  // DCIMVAC
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 10 && opc2 == 1) ||    // DCCMVAC
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 14 && opc2 == 1)) then // DCCIMVAC
  trap = write && HCR.TPC == '1';

elseif (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 1) ||  // ICIMVAU
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 0) ||     // ICIALLU
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 1 && opc2 == 0) ||     // ICIALLUIS
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 11 && opc2 == 1)) then // DCCMVAU
  trap = write && HCR.TPU == '1';

elseif (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 1) ||   // ACTLR
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 3)) then   // ACTLR2
  trap = HCR.TAC == '1';

elseif (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 2) ||  // TCMTR
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 3) ||     // TLBTR
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 6) ||     // REVIDR
  (op == SystemAccessType_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 7)) then  // AIDR
  trap = HCR.TID1 == '1';

elseif (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 1) ||               // ID_*
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 2 && opc2 <= 7) ||     // ID_*
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm >= 3 && opc2 <= 1) ||     // Reserved
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 3 && opc2 == 2) ||     // Reserved
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 IN {4,5})) then  // Reserved
  trap = HCR.TID3 == '1';

elseif op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 2 then   // CPACR
  trap = HCPTR.TCPAC == '1';

elseif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0 then  // PMCR
  trap = HDCR.TPMCR == '1' || HDCR.TPM == '1';

elseif (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8) ||      // PMEVCNTR<n>/PMEVTYPER<n>
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm IN {12,13,14}) || // PM*
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm IN {12,13,14}) ||  // PMCCNTR (MRRC/MCCR)
  (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm >= 4) then // PMCCNTR (MRRC/MCCR)
  trap = HDCR.TPM == '1';

elseif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then      // CNTP_TVAL CNTP_CTL CNTP_CVAL
  trap = CNTHCTL.PL1PCEN == '0';

elseif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then      // CNTP_TVAL CNTP_CTL CNTP_CVAL
  trap = CNTHCTL.PL1PCTEN == '0';

elseif (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then      // CNTP_TVAL CNTP_CTL CNTP_CVAL
  trap = CNTHCTL.PL1PCTEN == '0';

elseif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then      // CNTP_TVAL CNTP_CTL CNTP_CVAL
  trap = CNTHCTL.PL1PCTEN == '0';

elseif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then      // CNTP_TVAL CNTP_CTL CNTP_CVAL
  trap = CNTHCTL.PL1PCTEN == '0';

elseif (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then      // CNTP_TVAL CNTP_CTL CNTP_CVAL
  trap = CNTHCTL.PL1PCTEN == '0';

if trap then
  AArch32.AArch32SystemAccessTrap(EL2, instr);
// AArch32.CheckSystemAccessTraps()  
// Check for configurable disables or traps to a higher EL of an System register access.
AArch32.CheckSystemAccessTraps(bits(32) instr)
  if PSTATE.EL == EL0 then
    AArch32.CheckSystemAccessEL1Traps(instr);
  if EL2Enabled() && PSTATE.EL IN {EL0, EL1, EL2} && !IsInHost() then
    AArch32.CheckSystemAccessEL2Traps(instr);
  if HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.EL IN {EL0, EL1, EL2} then
    AArch64.CheckAArch32SystemAccessEL3Traps(instr);

// AArch32.DecodeSysRegAccess()  
// Decode an AArch32 System register access instruction into its operands.
(SystemAccessType, integer, integer, integer, integer, integer, boolean) AArch32.DecodeSysRegAccess(bits(32) instr)
  cp_num = UInt(instr<11:8>);
  // Decode the AArch32 System register access instruction
  if instr<31:28> !='1111' && instr<27:24> == '1110' && instr<4> == '1' then   // MRC/MCR
    op = SystemAccessType_RT;
    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn = UInt(instr<19:16>);
    CRm = UInt(instr<3:0>);
    write = instr<20> == '0';
  elsif instr<31:28> != '1111' && instr<27:21> == '1100010' then                // MRRC/MCRR
    op = SystemAccessType_RRT;
    opc1 = UInt(instr<7:4>);
    CRm = UInt(instr<3:0>);
    write = instr<20> == '0';
  elsif instr<31:28> != '1111' && instr<27:25> == '110' then                    // LDC/STC
    op = SystemAccessType_DT;
    CRn = UInt(instr<15:12>);
    write = instr<20> == '0';
  return (op, cp_num, opc1, CRn, CRm, opc2, write);

// Decodes an accepted access to a debug System register in the CP14 encoding space.  
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14DebugInstrDecode(bits(32) instr);

// Decodes an accepted access to a Jazelle System register in the CP14 encoding space.  
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14JazelleInstrDecode(bits(32) instr);

// Decodes an accepted access to a trace System register in the CP14 encoding space.  
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14TraceInstrDecode(bits(32) instr);
Library pseudocode for arch32/functions/coproc/CP15InstrDecode

// Decodes an accepted access to a System register in the CP15 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP15InstrDecode(bits(32) instr);

Library pseudocode for arch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

// AArch32.ExclusiveMonitorsPass()
// ----------------------------------

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.
boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.
acctype = AccType_ATOMIC;
iswrite = TRUE;
aligned = (address == Align(address, size));

if !aligned then
  secondstage = FALSE;
  AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
if !passed then
  return FALSE;
memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch32.Abort(address, memaddrdesc.fault);

passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
if passed then
  ClearExclusiveLocal(ProcessorID());
  if memaddrdesc.memattrs.shareable then
    passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

return passed;

Library pseudocode for arch32/functions/exclusive/AArch32.IsExclusiveVA

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
// It is always safe to return TRUE which will check the physical address only.
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);

Library pseudocode for arch32/functions/exclusive/AArch32.MarkExclusiveVA

// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
AArch32.SetExclusiveMonitors

```c
AArch32.SetExclusiveMonitors(bits(32) address, integer size)

acctype = AccType_ATOMIC;
iswrite = FALSE;
aligned = (address == Align(address, size));
memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  return;

if memaddrdesc.memattrs.shareable then
  MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
  MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
  AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

CheckAdvSIMDEnabled

```c
CheckAdvSIMDEnabled()

fpexc_check = TRUE;
advsimd = TRUE;
AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
```

CheckAdvSIMDOrVFPEnabled

```c
CheckAdvSIMDOrVFPEnabled(boolea include_fpexc_check, boolea advsimd)
AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
```

CheckCryptoEnabled32

```c
CheckCryptoEnabled32()
```
Library pseudocode for aarch32/functions/float/CheckVFPEnabled

```c
// CheckVFPEnabled()
// =================
CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted return;
```

Library pseudocode for aarch32/functions/float/FPHalvedSub

```c
// FPHalvedSub()
// =============
bits(N) FPHalvedSub(bits(N) op1, bits(N) op2,
                    FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity); inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero); zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/FPRSqrtStep

```c
// FPRSqrtStep()
// =============
bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity); inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero); zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0');
        result = FPHalvedSub(three, product, fpcr);
    return result;
```
Library pseudocode for aarch32/functions/float/FPRecipStep

```c
// FPRecipStep()
// =============

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
assert N IN {16,32};
FPCRType fpcr = StandardFPSCRValue();
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
  bits(N) product;
  if (inf1 && zero2) || (zero1 && inf2) then
    product = FPZero('0');
  else
    product = FPMul(op1, op2, fpcr);
  bits(N) two = FPTwo('0');
  result = FPSub(two, product, fpcr);
return result;
```

Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```c
// StandardFPSCRValue()
// ================

FPCRType StandardFPSCRValue()
return '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '0000000000000000000';
```

Library pseudocode for aarch32/functions/memory/AArch32.CheckAlignment

```c
// AArch32.CheckAlignment()
// ================

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType accctype, boolean iswrite)
if PSTATE.EL == EL0 && !ELUsingAArch32(S1TranslationRegime()) then
  A = SCTLR.A; //use AArch64 register, when higher Exception level is using AArch64
elsif PSTATE.EL == EL2 then
  A = HSCTLR.A;
else
  A = SCTLR.A;
aligned = (address == Align(address, alignment));
atomic = accctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMICRW, AccType_VEC};
ordered = accctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMICRW, AccType_VEC};
// AccType_VEC is used for SIMD element alignment checks only
check = (atomic || ordered || vector || A == '1');
if check && !aligned then
  secondstage = FALSE;
  AArch32.Abort(address, AArch32.AlignmentFault(accctype, iswrite, secondstage));
return aligned;
```
// AArch32.MemSingle[] - non-assignment (read) form
// -------------------------------------------------------------
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle{bits(32) address, integer size, AccType acctype, boolean wasaligned} = bits(size*8) value;
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
bits(size*8) value;
iswrite = FALSE;

// MMU or MPU
memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch32.Abort(address, memaddrdesc.fault);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
  if AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
    bits(4) ptag = TransformTag(ZeroExtend(address, 64));
    if !CheckTag(memaddrdesc, ptag, iswrite) then
      TagCheckFail(ZeroExtend(address, 64), iswrite);
  value = _Mem[memaddrdesc, size, accdesc];
return value;

// AArch32.MemSingle[] - assignment (write) form
// -------------------------------------------------------------
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle{bits(32) address, integer size, AccType acctype, boolean wasaligned} = bits(size*8) value;
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;

// MMU or MPU
memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch32.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareable then
  ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
  if AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
    bits(4) ptag = TransformTag(ZeroExtend(address, 64));
    if !CheckTag(memaddrdesc, ptag, iswrite) then
      TagCheckFail(ZeroExtend(address, 64), iswrite);
  _Mem[memaddrdesc, size, accdesc] = value;
return;
// AddressWithAllocationTag()
// ==========================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag = allocation_tag - ('000':address<55>);
    result<59:56> = tag;
    return result;

// AllocationTagFromAddress()
// ==========================
// Generate a Tag from a 64-bit value containing a Logical Address Tag.
// If access to Allocation Tags is disabled, this function returns '0000'.

bits(4) AllocationTagFromAddress(bits(64) tagged_address)
    bits(4) logical_tag = tagged_address<59:56>;
    bits(4) tag = logical_tag + ('000':tagged_address<55>);
    return tag;

// CheckTag()
// ===========
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean CheckTag(AddressDescriptor memaddrdesc, bits(4) ptag, boolean write)
    if memaddrdesc.memattrs.tagged then
        bits(64) paddress = ZeroExtend(memaddrdesc.paddress.address);
        return ptag == MemTag[paddress];
    else
        return TRUE;

// Hint_PreloadData
// ================

Hint_PreloadData(bits(32) address);

// Hint_PreloadDataForWrite
// ==============

Hint_PreloadDataForWrite(bits(32) address);

// Hint_PreloadInstr
// ===============

Hint_PreloadInstr(bits(32) address);
Library pseudocode for aarch32/functions/memory/MemA

```c
// MemA[] - non-assignment form
// ---------------------------------

bits(8*size) MemA[bits(32) address, integer size]
acctype = AccType_ATOMIC;
return Mem_with_type[address, size, acctype];

// MemA[] - assignment form
// ---------------------------------

MemA[bits(32) address, integer size] = bits(8*size) value
acctype = AccType_ATOMIC;
Mem_with_type[address, size, acctype] = value;
return;
```

Library pseudocode for aarch32/functions/memory/MemO

```c
// MemO[] - non-assignment form
// ---------------------------------

bits(8*size) MemO[bits(32) address, integer size]
acctype = AccType_ORDERED;
return Mem_with_type[address, size, acctype];

// MemO[] - assignment form
// ---------------------------------

MemO[bits(32) address, integer size] = bits(8*size) value
acctype = AccType_ORDERED;
Mem_with_type[address, size, acctype] = value;
return;
```
// MemTag[] - non-assignment (read) form
// ----------------------------------------
// Load an Allocation Tag from memory.

bits(4) MemTag[bits(64) address] = memaddrdesc;

 bits(4) value;
iswrite = FALSE;

 memaddrdesc = AArch64.TranslateAddress(address, AccType_NORMAL, iswrite, TRUE, TAG_GRANULE);

 // Check for aborts or debug exceptions
 if IsFault(memaddrdesc) then
   AArch64.Abort(address, memaddrdesc.fault);
 else
   return _MemTag[memaddrdesc];

// MemTag[] - assignment (write) form
// ----------------------------------------
// Store an Allocation Tag to memory.

MemTag[bits(64) address] = bits(4) value

 AddressDescriptor memaddrdesc;
iswrite = TRUE;

 // Stores of allocation tags must be aligned
 if address != Align(address, TAG_GRANULE) then
   boolean secondstage = FALSE;
   AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, iswrite, secondstage));

 wasaligned = TRUE;
 memaddrdesc = AArch64.TranslateAddress(address, AccType_NORMAL, iswrite, wasaligned, TAG_GRANULE);

 // Check for aborts or debug exceptions
 if IsFault(memaddrdesc) then
   AArch64.Abort(address, memaddrdesc.fault);

 // Memory array access
 if AllocationTagAccessIsEnabled() then
   _MemTag[memaddrdesc] = value;

// MemU[] - non-assignment form
// -------------------------------

 bits(8*size) MemU[bits(32) address, integer size] = Acctype;
 return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
// -------------------------------

MemU[bits(32) address, integer size] = bits(8*size) value

 acctype = AccType_NORMAL;
 Mem_with_type[address, size, acctype] = value;
 return;
Library pseudocode for aarch32/functions/memory/MemU_unpriv

// MemU_unpriv[] - non-assignment form
// ----------------------------------------

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType_UNPRIV;
    return Mem_with_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// ----------------------------------------

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_UNPRIV;
    Mem_with_type[address, size, acctype] = value;
    return;
// Mem_with_type[] - non-assignment (read) form
// -------------------------------------------
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
assert size IN {1, 2, 4, 8, 16};
bits(size*8) value;
boolean iswrite = FALSE;
aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
if !aligned then
  assert size > 1;
  value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

  // For subsequent bytes it is UNPREDICTABLE whether an unaligned Device memory access will generate an Alignment Fault, as to get this far means the first byte did not, so we must be changing to a new translation page.
  c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
  assert c IN {Constraint_FAULT, Constraint_NONE};
  if c == Constraint_NONE then aligned = TRUE;

  for i = 1 to size-1
    value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
  else
    value = AArch32.MemSingle[address, size, acctype, aligned];

  if BigEndian() then
    value = BigEndianReverse(value);
  return value;

// Mem_with_type[] - assignment (write) form
// ----------------------------------------
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
boolean iswrite = TRUE;
if BigEndian() then
  value = BigEndianReverse(value);
aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
if !aligned then
  assert size > 1;
  AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

  // For subsequent bytes it is UNPREDICTABLE whether an unaligned Device memory access will generate an Alignment Fault, as to get this far means the first byte did not, so we must be changing to a new translation page.
  c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
  assert c IN {Constraint_FAULT, Constraint_NONE};
  if c == Constraint_NONE then aligned = TRUE;

  for i = 1 to size-1
    AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
  else
    AArch32.MemSingle[address, size, acctype, aligned] = value;
return;
Library pseudocode for aarch32/functions/memory/TransformTag

// TransformTag()
// ==============
// Apply tag transformation rules.

define TransformTag(bits(64) vaddr)
    bits(4) vtag = vaddr<59:56>;
    bits(4) tagdelta = ZeroExtend(vaddr<55>);
    bits(4) ptag = vtag + tagdelta;
    return ptag;

Library pseudocode for aarch32/functions/memory/boolean

// boolean AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

define AccessIsTagChecked(bits(64) vaddr, AccType acctype)
    if PSTATE.M<4> == '1' then return FALSE;
    if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
        return FALSE;
    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') then
        return FALSE;
    if !AllocationTagAccessIsEnabled() then
        return FALSE;
    if acctype IN {AccType_IFETCH, AccType_PTW} then
        return FALSE;
    if acctype == AccType_NV2REGISTER then
        return FALSE;
    if PSTATE.TCO=='1' then
        return FALSE;
    if IsNonTagCheckedInstruction() then
        return FALSE;
    return TRUE;
// AArch32.ESBOperation()
// ---------------------
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

// Check if routed to AArch64 state
if !route_to_aarch64 && !ELUsingAArch32(EL1) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.EA == '1';
if route_to_aarch64 then
    AArch64.ESBOperation();
    return;

route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.EA == '1';
route_to_hyp = EL2Enabled() && PSTATE.EL IN {EL0, EL1} && (HCR.TGE == '1' || HCR.AMO == '1');
if route_to_monitor then
    target = M32_Monitor;
elsif route_to_hyp || PSTATE.M == M32_Hyp then
    target = M32_Hyp;
else
    target = M32_Abort;

if IsSecure() then
    mask_active = TRUE;
elsif target == M32_Monitor then
    mask_active = SCR.AW == '1' && (!HaveEL(EL2) || (HCR.TGE == '0' && HCR.AMO == '0'));
else
    mask_active = target == M32_Abort || PSTATE.M == M32_Hyp;
mask_set = PSTATE.A == '1';
(-, el) = ELFromM32(target);
intdis = Halted() || ExternalDebugInterruptsDisabled(el);
masked = intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending
if IsPhysicalSErrorPending() && masked then
    syndrome32 = AArch32.PhysicalSErrorSyndrome();
    DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
    ClearPendingPhysicalSError();
return;

// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();
// AArch32.ReportDeferredSError()
// -----------------------------
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredSError(bits(2) AET, bit ExT)
bits(32) target;
target<31> = '1'; // A
syndrome = Zeros(16);
if PSTATE.EL == EL2 then
    syndrome<11:10> = AET; // AET
    syndrome<9>     = ExT; // EA
    syndrome<5:0>   = '010001'; // DFSC
else
    syndrome<15:14> = AET; // AET
    syndrome<12>    = ExT; // ExT
    syndrome<9>     = TTBCR.EAE; // LPAE
    if TTBCR.EAE == '1' then
        syndrome<5:0> = '010001'; // STATUS
    else
        syndrome<10,3:0> = '10110'; // FS
if HaveAnyAArch64() then
    target<24:0> = ZeroExtend(syndrome); // Any RES0 fields must be set to zero
else
    target<15:0> = syndrome;
return target;

Library pseudocode for aarch32/functions/ras/AArch32.SErrorSyndrome

type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT )

Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

// AArch32.vESBOperation()
// ------------------------
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
assert EL2Enabled() && PSTATE.EL IN {EL0, EL1};

// Check for EL2 using AArch64 state
if !ELUsingAArch32(EL2) then
    AArch64.vESBOperation();
return;

// If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
// virtual SError interrupt might be pending
vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
vSEI_pending = vSEI_enabled && HCR.VA == '1';
vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
vmasked      = vientdis || PSTATE.A == '1';

// Check for a masked virtual SError pending
if vSEI_pending && vmasked then
    VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
    HCR.VA = '0'; // Clear pending virtual SError
return;
Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

```c
// AArch32.ResetGeneralRegisters()
// ---------------------------------
AArch32.ResetGeneralRegisters()
for i = 0 to 7
    R[i] = bits(32) UNKNOWN;
for i = 8 to 12
    Rmode[i, M32_User] = bits(32) UNKNOWN;
    Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
if HaveEL(EL2) then
    Rmode[13, M32_Hyp] = bits(32) UNKNOWN; // No R14_hyp
for i = 13 to 14
    Rmode[i, M32_User] = bits(32) UNKNOWN;
    Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
    Rmode[i, M32_Svc] = bits(32) UNKNOWN;
    Rmode[i, M32_Abt] = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        Rmode[i, M32_Monitor] = bits(32) UNKNOWN;
return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```c
// AArch32.ResetSIMDFPRegisters()
// --------------------------------
AArch32.ResetSIMDFPRegisters()
for i = 0 to 15
    Q[i] = bits(128) UNKNOWN;
return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

```c
// AArch32.ResetSpecialRegisters()
// --------------------------------
AArch32.ResetSpecialRegisters()
    // AArch32 special registers
    SPSR_fiq = bits(32) UNKNOWN;
    SPSR_irq = bits(32) UNKNOWN;
    SPSR_svc = bits(32) UNKNOWN;
    SPSR_abt = bits(32) UNKNOWN;
    SPSR_und = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;
    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```c
AArch32.ResetSystemRegisters(boolean cold_reset);
```
Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```plaintext
// ALUExceptionReturn()
// ==============

ALUExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User, M32_System} then
        UNPREDICTABLE; // UNDEFINED or NOP
    else
        AArch32.ExceptionReturn(address, SPSR[]);
```

Library pseudocode for aarch32/functions/registers/ALUWritePC

```plaintext
// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address, BranchType_INDIR);
    else
        BranchWritePC(address, BranchType_INDIR);
```

Library pseudocode for aarch32/functions/registers/BXWritePC

```plaintext
// BXWritePC()
// ===========

BXWritePC(bits(32) address, BranchType branch_type)
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the processor takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address<1> == '1' && ConstrainUnpredictableBool(Unpredictable_A32FORCEALIGNPC) then
            address<1> = '0';
        BranchTo(address, branch_type);
```

Library pseudocode for aarch32/functions/registers/BranchWritePC

```plaintext
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address, BranchType branch_type)
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
        BranchTo(address, branch_type);
```
Library pseudocode for aarch32/functions/registers/D

```c
// D[] - non-assignment form
// =========================

bits(64) D[integer n]
assert n >= 0 && n <= 31;
base = (n MOD 2) * 64;
return _V[n DIV 2]<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
assert n >= 0 && n <= 31;
base = (n MOD 2) * 64;
_V[n DIV 2]<base+63:base> = value;
return;
```

Library pseudocode for aarch32/functions/registers/Din

```c
// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
assert n >= 0 && n <= 31;
return _Dclone[n];
```

Library pseudocode for aarch32/functions/registers/LR

```c
// LR - assignment form
// ====================

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// ========================

bits(32) LR
    return R[14];
```

Library pseudocode for aarch32/functions/registers/LoadWritePC

```c
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType_INDIR);
```
Library pseudocode for aarch32/functions/registers/LookUpRIndex

```c
// LookUpRIndex()
// ==============

integer LookUpRIndex(integer n, bits(5) mode)
assert n >= 0 && n <= 14;

case n of // Select index by mode:   usr fiq irq svc abt und hyp
when 8     result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
when 9     result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
otherwise  result = n;

return result;
```

Library pseudocode for aarch32/functions/registers/Monitor_mode_registers

```c
bits(32) SP_mon;
bits(32) LR_mon;
```

Library pseudocode for aarch32/functions/registers/PC

```c
// PC - non-assignment form
// ===============

bits(32) PC

return R[15]; // This includes the offset from AArch32 state
```

Library pseudocode for aarch32/functions/registers/PCStoreValue

```c
// PCStoreValue()
// ==============

bits(32) PCStoreValue()

// This function returns the PC value. On architecture versions before ARMv7, it
// is permitted to instead return PC+4, provided it does so consistently. It is
// used only to describe A32 instructions, so it returns the address of the current
// instruction plus 8 (normally) or 12 (when the alternative is permitted).

return PC;
```

Library pseudocode for aarch32/functions/registers/Q

```c
// Q[] - non-assignment form
// ===============

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return _V[n];

// Q[] - assignment form
// ===============

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    _V[n] = value;
    return;
```
Library pseudocode for aarch32/functions/registers/Qin

```c
// Qin[] - non-assignment form
// -----------------------------------

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

Library pseudocode for aarch32/functions/registers/R

```c
// R[] - assignment form
// ----------------------

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// -------------------------

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

Library pseudocode for aarch32/functions/registers/RBankSelect

```c
// RBankSelect()
// ------------

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
    integer svc, integer abt, integer und, integer hyp)

case mode of
    when M32_User result = usr; // User mode
    when M32_FIQ result = fiq; // FIQ mode
    when M32_IRQ result = irq; // IRQ mode
    when M32_IRQ result = svc; // Supervisor mode
    when M32_Abort result = abt; // Abort mode
    when M32_Hyp result = hyp; // Hyp mode
    when M32_Undef result = und; // Undefined mode
    when M32_System result = usr; // System mode uses User mode registers
    otherwise Unreachable(); // Monitor mode

return result;
```
Library pseudocode for aarch32/functions/registers/Rmode

// Rmode[] - non-assignment form
// ---------------------------------

bits(32) Rmode(integer n, bits(5) mode)
assert n >= 0 && n <= 14;

// Check for attempted use of Monitor mode in Non-secure state.
if !IsSecure() then assert mode != M32_Monitor;
assert !BadMode(mode);

if mode == M32_Monitor then
    if n == 13 then return SP_mon;
    elsif n == 14 then return LR_mon;
    else return _R[LookUpRIndex(n, mode)]<31:0>;
else
    return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// ------------------------

Rmode(integer n, bits(5) mode) = bits(32) value
assert n >= 0 && n <= 14;

// Check for attempted use of Monitor mode in Non-secure state.
if !IsSecure() then assert mode != M32_Monitor;
assert !BadMode(mode);

if mode == M32_Monitor then
    if n == 13 then SP_mon = value;
    elsif n == 14 then LR_mon = value;
    else _R[LookUpRIndex(n, mode)]<31:0> = value;
else
    // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the
    // register are unchanged or set to zero. This is also tested for on
    // exception entry, as this applies to all AArch32 registers.
    if !HighestELUsingAArch32() && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
        _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
    else
        _R[LookUpRIndex(n, mode)]<31:0> = value;
    return;

Library pseudocode for aarch32/functions/registers/S

// S[] - non-assignment form
// ------------------------

bits(32) S[integer n]
assert n >= 0 && n <= 31;
base = (n MOD 4) * 32;
return _V[n DIV 4]<base+31:base>;

// S[] - assignment form
// ---------------------

S[integer n] = bits(32) value
assert n >= 0 && n <= 31;
base = (n MOD 4) * 32;
_V[n DIV 4]<base+31:base> = value;
return;
// SP - assignment form
// ====================
SP = bits(32) value
R[13] = value;
return;

// SP - non-assignment form
// ========================
bits(32) SP
return R[13];

array bits(64) _Dclone[0..31];

// AArch32.ExceptionReturn()
// =========================
AArch32.ExceptionReturn(bits(32) new_pc, bits(32) spsr)

SynchronizeContext();

// Attempts to change to an illegal mode or state will invoke the Illegal Execution state
// mechanism
SetPSTATEFromPSR(spsr);
ClearExclusiveLocal(ProcessorID());
SendEventLocal();
if PSTATE.IL == '1' then
   // If the exception return is illegal, PC[1:0] are UNKNOWN
   new_pc<1:0> = bits(2) UNKNOWN;
else
   // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
   if PSTATE.T == '1' then
      new_pc<0> = '0';                 // T32
   else
      new_pc<1:0> = '00';              // A32
BranchTo(new_pc, BranchType_ERET);

// AArch32.ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1CPR/WP
boolean AArch32.ExecutingATS1xPInstr()

if !HavePrivATExt() then return FALSE;

instr = ThisInstr();
if instr<24+:4> == '1110' && instr<8+:4> == '1110' then
   op1 = instr<21+:3>;
   CRn = instr<16+:4>;
   CRm = instr<0+:4>;
   op2 = instr<5+:3>;
   return (op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'001', '001'});
else
   return FALSE;
// AArch32.ExecutingCP10or11Instr()
// ================================================

boolean AArch32.ExecutingCP10or11Instr()
{  
    instr = ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
    {  
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
    }
    else // InstrSet_T32
    {  
        return (instr<31:28> == '111x' && (instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
    }
}

// AArch32.ExecutingLSMInstr()
// ===========================
// Returns TRUE if processor is executing a Load/Store Multiple instruction

boolean AArch32.ExecutingLSMInstr()
{  
    instr = ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
    {  
        return (instr<28+:4> != '1111' && instr<25+:3> == '100');
    }
    else // InstrSet_T32
    {  
        if ThisInstrLength() == 16 then
            return (instr<12+:4> == '1100');
        else
            return (instr<25+:7> == '1110100' && instr<22> == '0');
    }
}

// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
{  
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
}

// Read from a 32-bit AArch32 System register and return the register's contents.
bits(32) AArch32.SysRegRead(integer cp_num, bits(32) instr);

// Read from a 64-bit AArch32 System register and return the register's contents.
bits(64) AArch32.SysRegRead64(integer cp_num, bits(32) instr);
// AArch32.SysRegReadCanWriteAPSR()
// ------------------------------------------------------
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDCRint
        return TRUE;
    return FALSE;

// Write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, bits(32) val);

// Write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, bits(64) val);

// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.
AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M  = mode;
    PSTATE.EL = el;
    PSTATE.nRW = '1';
    PSTATE.SP  = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;
Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```c
// AArch32.WriteModeByInstr()
// ==========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.
AArch32.WriteModeByInstr(bits(5) mode)
valid, el = ELFromM32(mode);

// 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
// of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
// PSTATE.EL if it would result in any of:
// * A change to a mode that would cause entry to a higher Exception level.
if UInt(el) > UInt(PSTATE.EL) then
valid = FALSE;

// * A change to or from Hyp mode.
if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
valid = FALSE;

// * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
if PSTATE.M == M32_Monitor && HaveAArch32EL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
valid = FALSE;
if !valid then
PSTATE.IL = '1';
else
AArch32.WriteMode(mode);
```

Library pseudocode for aarch32/functions/system/BadMode

```c
// BadMode()
// =========

boolean BadMode(bits(5) mode)
// Return TRUE if 'mode' encodes a mode that is not valid for this implementation

// case mode of
when M32_Monitor
valid = HaveAArch32EL(EL3);
when M32_Hyp
valid = HaveAArch32EL(EL2);
when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
// If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
// state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
// AArch64, then these modes are EL1 modes.
// Therefore it is sufficient to test this implementation supports EL1 using AArch32.
valid = HaveAArch32EL(EL1);
when M32_User
valid = HaveAArch32EL(EL0);
otherwise
valid = FALSE; // Passed an illegal mode value

return !valid;
```
/ Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

// BankedRegisterAccessValid()
// -------------------------------------------
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

    case SYSm of
        when '000xx', '00100'                          // R8_usr to R12_usr
            if mode != M32_FIQ then UNPREDICTABLE;

        when '00101'                                   // SP_usr
            if mode == M32_System then UNPREDICTABLE;

        when '00110'                                   // LR_usr
            if mode IN {M32_Hyp, M32_System} then UNPREDICTABLE;

        when '010xx', '0110x', '01110'                 // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;

        when '1000x'                                   // LR_irq, SP_irq
            if mode == M32_IRQ then UNPREDICTABLE;

        when '1001x'                                   // LR_svc, SP_svc
            if mode == M32_Svc then UNPREDICTABLE;

        when '1010x'                                   // LR_abt, SP_abt
            if mode == M32_Abort then UNPREDICTABLE;

        when '1011x'                                   // LR_und, SP_und
            if mode == M32_Undef then UNPREDICTABLE;

        when '1110x'                                   // LR_mon, SP_mon
            if !HaveEL(EL2) || !IsSecure() || mode == M32_Monitor then UNPREDICTABLE;

        when '11110'                                   // ELR_hyp, only from Monitor or Hyp mode
            if !HaveEL(EL2) || !(mode IN {M32_Monitor, M32_Hyp}) then UNPREDICTABLE;

        when '11111'                                   // SP_hyp, only from Monitor mode
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;

        otherwise
            UNPREDICTABLE;

    return;
// CPSRWriteByInstr()
// ==================

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
privileged = PSTATE.EL != EL0; // PSTATE.<A,I,F,M> are not writable at EL0

// Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
if bytemask<3> == '1' then
    PSTATE.<N,Z,C,V,Q> = value<31:27>; // Bits <26:24> are ignored

if bytemask<2> == '1' then
    if privileged then
        PSTATE.PAN = value<22>; // Bits <21:20> are RES0
        PSTATE.GE = value<19:16>;
    if bytemask<1> == '1' then
        if privileged then
            PSTATE.E = value<9>; // PSTATE.E is writable at EL0
            if privileged then
                PSTATE.A = value<8>;
        if bytemask<0> == '1' then
            if privileged then
                PSTATE.<I,F> = value<7:6>; // Bit <5> is RES0
                if CurrentInstrSet() == InstrSet_T32 then
                    AArch32.WriteModeByInstr(value<4:0>);
                return;

Library pseudocode for aarch32/functions/system/ConditionPassed

// ConditionPassed()
// ================

boolean ConditionPassed()
return ConditionHolds(AArch32.CurrentCond());

Library pseudocode for aarch32/functions/system/CurrentCond

bits(4) AArch32.CurrentCond();

Library pseudocode for aarch32/functions/system/InITBlock

// InITBlock()
// ===========

boolean InITBlock()
if CurrentInstrSet() == InstrSet_T32 then
    return PSTATE.IT<3:0> != '0000';
else
    return FALSE;

Library pseudocode for aarch32/functions/system/LastInITBlock

// LastInITBlock()
// ===============

boolean LastInITBlock()
return (PSTATE.IT<3:0> == '1000');
Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```c
// SPSRWriteByInstr()  
// ================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)
new_spsr = SPSR[];
if bytemask<3> == '1' then
    new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits
if bytemask<2> == '1' then
if bytemask<1> == '1' then
    new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask
if bytemask<0> == '1' then
    new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits
SPSR[] = new_spsr;                   // UNPREDICTABLE if User or System mode
return;
```

Library pseudocode for aarch32/functions/system/SPSRaccessValid

```c
// SPSRaccessValid()  
// ================

// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110'                                                   // SPSR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '10000'                                                   // SPSR_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '10010'                                                   // SPSR_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '10100'                                                   // SPSR_abt
            if mode == M32_Abt then UNPREDICTABLE;
        when '10110'                                                   // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100'                                                   // SPSR_mon
            if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
        when '11110'                                                   // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;
    return;
```

Library pseudocode for aarch32/functions/system/SelectInstrSet

```c
// SelectInstrSet()  
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN (InstrSet_A32, InstrSet_T32);
    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';
    return;
```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;

Library pseudocode for aarch32/functions/v6simd/SignedSat

// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;

Library pseudocode for aarch32/functions/v6simd/UnsignedSat

// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB);
        TEX = texcb<4:2>;  C = texcb<1>;  B = texcb<0>;
    case TEX:C:B of
    when '00000'
        // Device-nGnRnE
        memattrs.type = MemType_Device;
        memattrs.device = DeviceType_nGnRnE;
    when '00001', '01000'
        // Device-nGnRE
        memattrs.type = MemType_Device;
        memattrs.device = DeviceType_nGnRE;
    when '00010', '00011', '00100'
        // Write-back or Write-through Read allocate, or Non-cacheable
        memattrs.type = MemType_Normal;
        memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
        memattrs.outer = ShortConvertAttrsHints(C:B, acctype, FALSE);
        memattrs.shareable = (S == '1');
    when '00110'
        memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    when '00111'
        // Write-back Read and Write allocate
        memattrs.type = MemType_Normal;
        memattrs.inner = ShortConvertAttrsHints('01', acctype, FALSE);
        memattrs.outer = ShortConvertAttrsHints('01', acctype, FALSE);
        memattrs.shareable = (S == '1');
    when '1xxxx'
        // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
        memattrs.type = MemType_Normal;
        memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
        memattrs.outer = ShortConvertAttrsHints(TEX<1:0>, acctype, FALSE);
        memattrs.shareable = (S == '1');
    otherwise
        // Reserved, handled above
        Unreachable();

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;

    // distinction between inner and outer shareable is not supported in this format
    memattrs.outershareable = memattrs.shareable;
    memattrs.tagged = FALSE;

    return MemAttrDefaults(memattrs);
// AArch32.InstructionDevice()
// ------------------------
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch32.InstructionDevice(AddressDescriptor addrsdesc, bits(32) vaddress,
                                          bits(40) ipaddress, integer level, bits(4) domain,
                                          AccType acctype, boolean iswrite, boolean secondstage,
                                          boolean s2fs1walk)

    c = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrsdesc.fault = AArch32.PermissionFault(ipaddress, domain, level, acctype, iswrite,
                                                secondstage, s2fs1walk);
    else
        addrsdesc.memattrs.type = MemType_Normal;
        addrsdesc.memattrs.inner.attrs = MemAttr_NC;
        addrsdesc.memattrs.inner.hints = MemHint_No;
        addrsdesc.memattrs.outer = addrsdesc.memattrs.inner;
        addrsdesc.memattrs.tagged = FALSE;
        addrsdesc.memattrs = MemAttrDefaults(addrsdesc.memattrs);
    end if

    return addrsdesc;
// AArch32.RemappedTEXDecode()
// ===========================

MemoryAttributes AArch32.RemappedTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;

    region = UInt(TEX<0>:C:B);  // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        base = 2 * region;
        attrfield = PRRR<base+1:base>;

        if attrfield == '11' then  // Reserved, maps to allocated value
            (-, attrfield) = ConstranUnpredictableBits(Unpredictable_RESPRR);

        case attrfield of
            when '00'                  // Device-nGnRnE
                memattrs.type = MemType_Device;
                memattrs.device = DeviceType_nGnRnE;
            when '01'                  // Device-nGnRE
                memattrs.type = MemType_Device;
                memattrs.device = DeviceType_nGnRE;
            when '10'
                memattrs.type = MemType_Normal;
                memattrs.inner = ShortConvertAttrsHints(NMRR<base+1:base>, acctype, FALSE);
                memattrs.outer = ShortConvertAttrsHints(NMRR<base+17:base+16>, acctype, FALSE);
                s_bit = if S == '0' then PRRR.NS0 else PRRR.NS1;
                memattrs.shareable = (s_bit == '1');
                memattrs.outershareable = (s_bit == '1' && PRRR<region+24> == '0');
            when '11'
                Unreachable();

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tagged = FALSE;

    return MemAttrDefaults(memattrs);
// AArch32.S1AttrDecode()
// ===============
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch32.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)
    MemoryAttributes memattrs;
    if PSTATE.EL == EL2 then
        mair = HMAIR1:HMAIR0;
    else
        mair = MAIR1:MAIR0;
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;
    memattrs.tagged = FALSE;
    if ((attrfield<7:4> != '0000' && attrfield<7:4> != '1111' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (_, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);
    if !HaveMTEExt() && attrfield<7:4> == '1111' && attrfield<3:0> == '0000' then
        // Reserved, maps to an allocated value
        (_, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);
    if attrfield<7:4> == '0000' then            // Device
        memattrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000' memattrs.device = DeviceType_nGnRnE;
            when '0100' memattrs.device = DeviceType_nGnRE;
            when '1000' memattrs.device = DeviceType_nGRE;
            when '1100' memattrs.device = DeviceType_GRE;
            otherwise Unreachable();         // Reserved, handled above
    elsif attrfield<3:0> != '0000' then        // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrsoutershareable = SH == '10';
    elsif HaveMTEExt() && attrfield == '11110000' then // Tagged, Normal
        memattrs.tagged = TRUE;
        memattrs.type = MemType_Normal;
        memattrs.outer.attrs = MemAttr WB;
        memattrs.inner.attrs = MemAttr WB;
        memattrs.outer.hints = MemHint_RWA;
        memattrsinner.hints = MemHint_RWA;
        memattrs.shareable = SH<1> == '1';
        memattrsoutershareable = SH == '10';
    else
        Unreachable();         // Reserved, handled above
    return MemAttrDefaults(memattrs);
// AArch32.TranslateAddressS1Off()
// =================================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch32.TranslateAddressS1Off(bits(32) vaddress, AccType acctype, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());

TLBRecord result;

default_cacheable = (HasS2Translation() && ((if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC) == '1'));

if default_cacheable then
  // Use default cacheable settings
  result.addrdesc.memattrs.type = MemType_Normal;
  result.addrdesc.memattrs.inner.attrs = MemAttr_WB;     // Write-back
  result.addrdesc.memattrs.inner.hints = MemHint_RWA;
  result.addrdesc.memattrs.shareable = FALSE;
  result.addrdesc.memattrsoutershareable = FALSE;
  result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
elsif acctype != AccType_IFETCH then
  // Treat data as Device
  result.addrdesc.memattrs.type = MemType_Device;
  result.addrdesc.memattrs.device = DeviceType_nGnRnE;
  result.addrdesc.memattrs.inner = MemAttrHints_UNKNOWN;
  result.addrdesc.memattrs.tagged = FALSE;
else
  // Instruction cacheability controlled by SCTLR/HSCTLR.I
  if PSTATE.EL == EL2 then
    cacheable = HSCTLRI == '1';
  else
    cacheable = SCTLR.I == '1';
  result.addrdesc.memattrs.type = MemType_Normal;
  if cacheable then
    result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
    result.addrdesc.memattrs.inner.hints = MemHint_RA;
  else
    result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
    result.addrdesc.memattrs.inner.hints = MemHint_No;
  result.addrdesc.memattrs.shareable = TRUE;
  result.addrdesc.memattrsoutershareable = TRUE;
  result.addrdesc.memattrs.tagged = FALSE;
result.addrdesc.memattrs.outer = result.addrdesc.memattrsinner;
result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);
result.perms.ap = bits(3) UNKNOWN;
result.perms.xn = '0';
result.perms.pxn = '0';
result.nG = bit UNKNOWN;
result.contiguous = boolean UNKNOWN;
result.domain = bits(4) UNKNOWN;
result.level = integer UNKNOWN;
result.blocksize = integer UNKNOWN;
result.addrdesc.paddress.address = ZeroExtend(vaddress);
result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
result.addrdesc.fault = AArch32.NoFault();
return result;
Library pseudocode for aarch32/translation/checks/AArch32.AccessIsPrivileged

    // AArch32.AccessIsPrivileged()
    // -----------------------------
    boolean AArch32.AccessIsPrivileged(AccType acctype)
      el = AArch32.AccessUsesEL(acctype);
      if el == EL0 then
        ispriv = FALSE;
      elsif el != EL1 then
        ispriv = TRUE;
      else
        ispriv = (acctype != AccType_UNPRIV);
      return ispriv;

Library pseudocode for aarch32/translation/checks/AArch32.AccessUsesEL

    // AArch32.AccessUsesEL()
    // ---------------------
    // Returns the Exception Level of the regime that will manage the translation for a given access type.
    bits(2) AArch32.AccessUsesEL(AccType acctype)
      if acctype == AccType_UNPRIV then
        return EL0;
      else
        return PSTATE.EL;

Library pseudocode for aarch32/translation/checks/AArch32.CheckDomain

    // AArch32.CheckDomain()
    // ----------------------
    (boolean, FaultRecord) AArch32.CheckDomain(bits(4) domain, bits(32) vaddress, integer level, AccType acctype, boolean iswrite)
      index = 2 * UInt(domain);
      attrfield = DACR<index+1:index>;
      if attrfield == '10' then          // Reserved, maps to an allocated value
        // Reserved value maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESDACR);
      if attrfield == '00' then
        fault = AArch32.DomainFault(domain, level, acctype, iswrite);
      else
        fault = AArch32.NoFault();
      permissioncheck = (attrfield == '01');
      return (permissioncheck, fault);
Library pseudocode for aarch32/translation/checks/AArch32.CheckPermission
AArch32.CheckPermission()
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level,
bits(4) domain, bit NS, AccType acctype, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());

if PSTATE.EL != EL2 then
wxn = SCTLR.WXN == '1';
if TTBCR.EAE == '1' || SCTLR.AFE == '1' || perms.ap<0> == '1' then
priv_r = TRUE;
priv_w = perms.ap<2> == '0';
user_r = perms.ap<1> == '1';
user_w = perms.ap<2:1> == '01';
else
priv_r = perms.ap<2:1> != '00';
priv_w = perms.ap<2:1> == '01';
user_r = perms.ap<1> == '1';
user_w = FALSE;
uwxn = SCTLR.UWXN == '1';
else
priv_r = perms.ap<2:1> != '00';
priv_w = perms.ap<2:1> == '01';
user_r = perms.ap<1> == '1';
user_w = FALSE;

ispriv = AArch32.AccessIsPrivileged(acctype);

pan = if HavePANExt() then PSTATE.PAN else '0';
is_ldst = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
is_atslxp = (acctype == AccType_AT && AArch32.ExecutingATS1xPInstr());
if pan == '1' && user_r && ispriv && (is_ldst || is_atslxp) then
priv_r = FALSE;
priv_w = FALSE;

user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
(priv_w && wxn) || (user_w && uwxn));
if ispriv then
   (r, w, xn) = (priv_r, priv_w, priv_xn);
else
   (r, w, xn) = (user_r, user_w, user_xn);
else
   // Access from EL2
wxn = HSCTLR.WXN == '1';
r = TRUE;
w = perms.ap<2> == '0';

// Restriction on Secure instruction fetch
if HaveEL(EL3) && IsSecure() && NS == '1' then
    secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
else
    secure_instr_fetch = '1' then xn = TRUE;

if acctype == AccType_IFETCH then
fail = xn;
failedread = TRUE;
elif acctype IN {AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDDATOMICRW} then
fail = !r || !w;
failedread = !r;
elif acctype == AccType_DC then
// DC maintenance instructions operating by VA, cannot fault from stage 1 translation.
fail = FALSE;
elif iswrite then
fail = !w;
failedread = FALSE;
else
fail = !r;
failedread = TRUE;
if fail then
secondstage = FALSE;
s2fs1walk = FALSE;
```
ipaddress = bits(40) UNKNOWN;
return AArch32.PermissionFault(ipaddress, domain, level, acctype, 
  !failedread, secondstage, s2fs1walk);
else
  return AArch32.NoFault();

Library pseudocode for aarch32/translation/checks/AArch32.CheckS2Permission

// AArch32.CheckS2Permission()
// -----------------------------------
// Function used for permission checking from AArch32 stage 2 translations

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(40) ipaddress, 
  integer level, AccType acctype, boolean iswrite, 
  boolean s2fs1walk)

assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

r = perms.ap<1> == '1';
w = perms.ap<2> == '1';
if HaveExtendedExecuteNeverExt() then
  case perms.xn:perms.xxn of
    when '00'  xn = !r;
    when '01'  xn = !r || PSTATE.EL == EL1;
    when '10'  xn = TRUE;
    when '11'  xn = !r || PSTATE.EL == EL0;
  else
    xn = !r || perms.xn == '1';
  // Stage 1 walk is checked as a read, regardless of the original type
  if acctype == AccType_IFETCH && !s2fs1walk then
    fail = xn;
    failedread = TRUE;
  elseif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW }) && !s2fs1walk then
    fail = !r || !w;
    failedread = !r;
  elseif acctype == AccType_DC && !s2fs1walk then
    // DC maintenance instructions operating by VA, do not generate Permission faults
    // from stage 2 translation, other than from stage 1 translation table walk.
    fail = FALSE;
  elseif iswrite && !s2fs1walk then
    fail = !w;
    failedread = FALSE;
  else
    fail = !r;
    failedread = !iswrite;
  if fail then
    domain = bits(4) UNKNOWN;
    secondstage = TRUE;
  return AArch32.PermissionFault(ipaddress, domain, level, acctype, 
    !failedread, secondstage, s2fs1walk);
else
  return AArch32.NoFault();
```
// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32 translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
assert ELUsingAArch32(S1TranslationRegime());
assert size IN {2,4};
match = FALSE;
mismatch = FALSE;
for i = 0 to UInt(DBGIDIR.BRPs)
    (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
match = match || match_i;
mismatch = mismatch || mismatch_i;
if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Breakpoint;
    Halt(reason);
elsif (match || mismatch) && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
    acctype = AccType_IFETCH;
iswrite = FALSE;
d debugmoe = DebugException_Breakpoint;
return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();
end

Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

// AArch32.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

FaultRecord fault = AArch32.NoFault();
d_side = (acctype != AccType_IFETCH);
generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
halt = HaltOnBreakpointOrWatchpoint();
// Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);
if !d_side && vector_catch_first && generate_exception then
    fault = AArch32.CheckVectorCatch(vaddress, size);
endif
if fault.type == Fault_None && (generate_exception || halt) then
    if d_side then
        fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
    else
        fault = AArch32.CheckBreakpoint(vaddress, size);
    endif
endif
if fault.type == Fault_None && !d_side && !vector_catch_first && generate_exception then
    return AArch32.CheckVectorCatch(vaddress, size);
endif
return fault;
Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// -------------------------
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// Vector Catch can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
assert ELUsingAArch32(S1TranslationRegime());

match = AArch32.VCRMatch(vaddress);
if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
    match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

if match && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    debugmoe = DebugException_VectorCatch;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// -------------------------
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                 boolean iswrite, integer size)
assert ELUsingAArch32(S1TranslationRegime());

match = FALSE;
ispriv = AArch32.AccessIsPrivileged(acctype);
for i = 0 to UInt(DBGDIDR.WRPs)
    match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Watchpoint;
    Halt(reason);
elsif match && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
    debugmoe = DebugException_Watchpoint;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();
```

Library pseudocode for aarch32/translation/faults/AArch32.AccessFlagFault

```
// AArch32.AccessFlagFault()
// -------------------------

FaultRecord AArch32.AccessFlagFault(bits(40) ipaddress, bits(4) domain, integer level,
                 AccType acctype, boolean iswrite, boolean secondstage,
                 boolean s2fs1walk)

extflag = bit UNKNOWN;
dbgmoe = bits(4) UNKNOWN;
errortype = bits(2) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_AccessFlag, ipaddress, domain, level, acctype, iswrite,
                 extflag, dbgmoe, errortype, secondstage, s2fs1walk);
```
Library pseudocode for aarch32/translation/faults/AArch32.AddressSizeFault

```plaintext
// AArch32.AddressSizeFault()
// =========================

FaultRecord AArch32.AddressSizeFault(bits(40) ipaddress, bits(4) domain, integer level,
AccType acctype, boolean iswrite, boolean secondstage,
boolean s2fs1walk)

extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
errtype = bits(2) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_AddressSize, ipaddress, domain, level, acctype, iswrite,
extflag, debugmoe, errrtype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.AlignmentFault

```plaintext
// AArch32.AlignmentFault()
// ========================

FaultRecord AArch32.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

ipaddress = bits(40) UNKNOWN;
domain = bits(4) UNKNOWN;
level = integer UNKNOWN;
extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
errtype = bits(2) UNKNOWN;
s2fs1walk = boolean UNKNOWN;
return AArch32.CreateFaultRecord(Fault_Alignment, ipaddress, domain, level, acctype, iswrite,
extflag, debugmoe, errrtype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.AsynchExternalAbort

```plaintext
// AArch32.AsynchExternalAbort()
// -----------------------------

// Wrapper function for asynchronous external aborts

FaultRecord AArch32.AsynchExternalAbort(boolean parity, bits(2) errrtype, bit extflag)

type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
ipaddress = bits(40) UNKNOWN;
domain = bits(4) UNKNOWN;
level = integer UNKNOWN;
acctype = AccType_NORMAL;
iswrite = boolean UNKNOWN;
debugmoe = bits(4) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;
return AArch32.CreateFaultRecord(type, ipaddress, domain, level, acctype, iswrite, extflag,
debugmoe, errrtype, secondstage, s2fs1walk);
```
Library pseudocode for aarch32/translation/faults/AArch32.DebugFault

```c
// AArch32.DebugFault()
// -------------------

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)

  ipaddress = bits(40) UNKNOWN;
  domain = bits(4) UNKNOWN;
  errortype = bits(2) UNKNOWN;
  level = integer UNKNOWN;
  extflag = bit UNKNOWN;
  secondstage = FALSE;
  s2fs1walk = FALSE;

  return AArch32.CreateFaultRecord(Fault_Debug, ipaddress, domain, level, acctype, iswrite,
                                   extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.DomainFault

```c
// AArch32.DomainFault()
// --------------------

FaultRecord AArch32.DomainFault(bits(4) domain, integer level, AccType acctype, boolean iswrite)

  ipaddress = bits(40) UNKNOWN;
  extflag = bit UNKNOWN;
  debugmoe = bits(4) UNKNOWN;
  errortype = bits(2) UNKNOWN;
  secondstage = FALSE;
  s2fs1walk = FALSE;

  return AArch32.CreateFaultRecord(Fault_Domain, ipaddress, domain, level, acctype, iswrite,
                                   extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.NoFault

```c
// AArch32.NoFault()
// ----------------

FaultRecord AArch32.NoFault()

  ipaddress = bits(40) UNKNOWN;
  domain = bits(4) UNKNOWN;
  level = integer UNKNOWN;
  acctype = AccType_NORMAL;
  iswrite = boolean UNKNOWN;
  extflag = bit UNKNOWN;
  debugmoe = bits(4) UNKNOWN;
  errortype = bits(2) UNKNOWN;
  secondstage = FALSE;
  s2fs1walk = FALSE;

  return AArch32.CreateFaultRecord(Fault_None, ipadress, domain, level, acctype, iswrite,
                                   extflag, debugmoe, errortype, secondstage, s2fs1walk);
```
// AArch32.PermissionFault()
// =========================

FaultRecord AArch32.PermissionFault(bits(40) ipaddress, bits(4) domain, integer level,
                                        AccType acctype, boolean iswrite, boolean secondstage,
                                        boolean s2fslwalk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault Permission, ipaddress, domain, level, acctype, iswrite,
                                      extflag, debugmoe, errortype, secondstage, s2fslwalk);

// AArch32.TranslationFault()
// ==========================

FaultRecord AArch32.TranslationFault(bits(40) ipaddress,  bits(4) domain, integer level,
                                        AccType acctype, boolean iswrite, boolean secondstage,
                                        boolean s2fslwalk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault Translation, ipaddress, domain, level, acctype, iswrite,
                                      extflag, debugmoe, errortype, secondstage, s2fslwalk);
// AArch32.FirstStageTranslate()
// --------------------------------
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.FirstStageTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
        boolean wasaligned, integer size)

    if PSTATE.EL == EL2 then
        s1_enabled = HSCTRL.M == '1';
    elsif EL2Enabled then
        tge = (if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE);
        dc = (if ELUsingAArch32(EL2) then HCR_DC else HCR_EL2_DC);
        s1_enabled = tge == '0' && dc == '0' && SCTLR.M == '1';
    else
        s1_enabled = SCTLR.M == '1';
    end

    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then // First stage enabled
        use_long_descriptor_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
        if use_long_descriptor_format then
            S1 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                s2fs1walk, size);
            permissioncheck = TRUE; domaincheck = FALSE;
        else
            S1 = AArch32.TranslationTableWalkSD(vaddress, acctype, iswrite, size);
            permissioncheck = TRUE; domaincheck = TRUE;
        end
    else
        S1 = AArch32.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE; domaincheck = FALSE;
    end

    if UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() then
        if S1.addrdesc.memattrs.type == MemType_Device && S1.addrdesc.memattrs.device != DeviceType_GRE
            nTLSMD = if S1TranslationRegime() == EL2 then HSCTRL.nTLSMD else SCTLR.nTLSMD;
            if nTLSMD == '0' then
                S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
            end
        end
    end

    // Check for unaligned data accesses to Device memory
    if (!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DC2VA) && S1.addrdesc.memattrs.type == MemType_Device && !IsFault(S1.addrdesc) then
        S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
    end

    if IsFault(S1.addrdesc) && domaincheck then
        (permissioncheck, abort) = AArch32.CheckDomain(S1.domain, vaddress, S1.level, acctype, iswrite);
        S1.addrdesc.fault = abort;
    end

    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch32.CheckPermission(S1.perms, vaddress, S1.level,
            S1.domain, S1.addrdesc.paddress.NS, acctype, iswrite);
    end

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device && acctype == AccType_IFETCH) then
        S1.addrdesc = AArch32.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
            S1.domain, acctype, iswrite, secondstage, s2fs1walk);
    end

    return S1.addrdesc;
// AArch32.FullTranslate()
// ----------------------
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch32.FullTranslate(bits(32) vaddress, AccType accype, boolean iswrite,
                                          boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch32.FirstStageTranslate(vaddress, accype, iswrite, wasaligned, size);
    if !IsFault(S1) && !(HaveNV2Ext() && accype == AccType_NV2REGISTER) && HasS2Translation() then
        s2fs1walk = FALSE;
        result = AArch32.SecondStageTranslate(S1, vaddress, accype, iswrite, wasaligned, s2fs1walk, size);
    else
        result = S1;
    return result;
// AArch32.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.SecondStageTranslate(AddressDescriptor S1, bits(32) vaddress,
        AccType acctype, boolean iswrite, boolean wasaligned,
        boolean s2fs1walk, integer size)

assert HasS2Translation();
assert IsZero(S1.paddress.address<47:40>);
hwupdatewalk = FALSE;
if !ELUsingAArch32(EL2) then
    return AArch64.SecondStageTranslate(S1, ZeroExtend(vaddress, 64), acctype, iswrite,
                                           wasaligned, s2fs1walk, size, hwupdatewalk);

s2_enabled = HCR.VM == '1' || HCR.DC == '1';
secondstage = TRUE;
if s2_enabled then // Second stage enabled
    ipaddress = S1.paddress.address<39:0>;
    S2 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);

    // Check for unaligned data accesses to Device memory
    if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
        && S2.addrdesc.memattrs.type == MemType_Device && !IsFault(S2.addrdesc) then
        S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

    // Check for permissions on Stage2 translations
    if !IsFault(S2.addrdesc) then
        S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                      acctype, iswrite, s2fs1walk);

    // Check for instruction fetches from Device memory not marked as execute-never. As there
    // has not been a Permission Fault then the memory is not marked execute-never.
    if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
        acctype == AccType_IFETCH) then
        domain = bits(4) UNKNOWN;
        S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                 domain, acctype, iswrite,
                                                 secondstage, s2fs1walk);

    // Check for protected table walk
    if (s2fs1walk && !IsFault(S2.addrdesc) && HCR.PTW == '1' &&
        S2.addrdesc.memattrs.type == MemType_Device) then
        domain = bits(4) UNKNOWN;
        S2.addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, S2.level, acctype,
                                                     iswrite, secondstage, s2fs1walk);
    result = CombineS1S2Desc(S1, S2.addrdesc);
else
    result = S1;
return result;
Library pseudocode for aarch32/translation/translation/AArch32.SecondStageWalk

// AArch32.SecondStageWalk()
// ================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch32.SecondStageWalk(AddressDescriptor S1, bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    assert HasS2Translation();

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk, size);

Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

// AArch32.TranslateAddress()
// ================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

    if !ELUsingAArch32(S1TranslationRegime()) then
        return AArch64.TranslateAddress(ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned, size);
    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);
    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);
    return result;
TLBRecord AArch32.TranslationTableWalkLD(bits(40) ipaddress, bits(32) vaddress, 
   AccType acctype, boolean iswrite, boolean secondstage, 
   boolean s2fs1walk, integer size)
if !secondstage then
   assert ELUsingAArch32(S1TranslationRegime());
else
   assert HaveEl(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();
TLBRecord result;
AddressDescriptor descaddr;
bits(64) baseregister;
bits(40) inputaddr;     // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
domain = bits(4) UNKNOWN;
descaddr.memattrs.type = MemType_Normal;

// Fixed parameters for the page table walk:
//  grainsize = Log2(Size of Table)         - Size of Table is 4KB in AArch32
//  stride = Log2(Address per Level)        - Bits of address consumed at each level
constant integer grainsize = 12;                    // Log2(4KB page size)
constant integer stride = grainsize - 3;            // Log2(page size / 8 bytes)

// Derived parameters for the page table walk:
//  inputsize = Log2(Size of Input Address) - Input Address size in bits
//  level = Level to start walk from
// This means that the number of levels after start level = 3-level
if !secondstage then
   // First stage translation
   inputaddr = ZeroExtend(vaddress);
el = AArch32.AccessUsesEL(acctype);
   if el == EL2 then
      inputsize = 32 - UInt(HTCR.T0SZ);
      basefound = inputsize == 32 || IsZero(inputaddr<31:inputsize>);
      disabled = FALSE;
      baseregister = HTTR;
      descaddr.memattrs = WalkAttrDecode(HTCR.SH0, HTCRR.ORGN0, HTCRR.IRGN0, secondstage);
      reversedescriptors = HSCTRL.EE == '1';
      lookupsecure = FALSE;
      singlepriv = TRUE;
      hierattrdisables = AArch32.HaveHPDExt() && HTCRR.HPD == '1';
   else
      basefound = FALSE;
      disabled = FALSE;
      t0size = UInt(TTBCR.T0SZ);
      if t0size == 0 || IsZero(inputaddr<31:(32-t0size)>) then
         inputsize = 32 - t0size;
         basefound = TRUE;
         baseregister = TTBR0;
      descaddr.memattrs = WalkAttrDecode(TTBCR.SH0, TTBCR.ORGN0, TTBCR.IRGN0, secondstage);
      hierattrdisables = AArch32.HaveHPDExt() && TTBCR.T2E == '1' && TTBCR2.HPD0 == '1';
      t1size = UInt(TTBCR.T1SZ);
   if (t1size == 0 && !basefound) || (t1size > 0 && IsOnes(inputaddr<31:(32-t1size)>)) then
      inputsize = 32 - t1size;
      basefound = TRUE;
      baseregister = TTBR1;
      descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGN1, TTBCR.IRGN1, secondstage);
      hierattrdisables = AArch32.HaveHPDExt() && TTBCR.T2E == '1' && TTBCR2.HPD1 == '1';
      reversedescriptors = SCTLR.EE == '1';
      lookupsecure = IsSecure();
      singlepriv = FALSE;
// The starting level is the number of strides needed to consume the input address
level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

else

// Second stage translation
inputaddr = ipaddress;
inputsize = 32 - SInt(VTCR.T0SZ);
// VTCR.S must match VTCR.T0SZ
if VTCR.S != VTCR.T0SZ
(-, inputsize) = ConstrainUnpredictableInteger(32-7, 32+8, Unpredictable_RESVTCRS);
basefound = inputsize == 40 || IsZero(inputaddr<39:inputsize>);
disabled = FALSE;
descaddr.memattrs = WalkAttrDecode(VTCR.IRGNO, VTCR.ORGNO, VTCR.SH0, secondstage);
reversedescriptors = HSCTLRT.EE == '1';
singlepriv = TRUE;

if !basefound || disabled then

level = 1;           // AArch64 reports this as a level 0 fault
result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
return result;

if !IsZero(baseregister<47:40>) then

level = 0;
result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
return result;

// Bottom bound of the Base address is:
// 2^((8 bytes per entry) + Log2(Number of entries in starting level table))
// Number of entries in starting level table =
// (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
baseaddress = baseregister<39:baselowerbound>:Zeros;

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsize - 1;
repeat

addrselectbottom = (3-level)*stride + grainsize;

bits(40) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
descaddr.paddress.address = ZeroExtend(baseaddress OR index);
descaddr.paddress.NS = ns_table;

// If there are two stages of translation, then the first stage table walk addresses
// are themselves subject to translation
if secondstage || !HasS2Translation() || (HaveNV2Ext() && acctype == AccType_NV2REGISTER) then
descaddr2 = descaddr;

Shared Pseudocode Functions
else
descaddr2 = AArch32.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
// Check for a fault on the stage 2 walk
if IsFault(descaddr2) then
    result.addrdesc.fault = descaddr2.fault;
    return result;

// Update virtual address for abort functions
descaddr2.vaddress = ZeroExtend(vaddress);
accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
desc = _Mem[descaddr2, 8, accdesc];
if reversedescriptors then desc = BigEndianReverse(desc);
if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
    // Fault (00), Reserved (10), or Block (01) at level 3.
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;

// Valid Block, Page, or Table entry
if desc<1:0> == '01' || level == 3 then       // Block (01) or Page (11)
    blocktranslate = TRUE;
else                                      // Table (11)
    if !IsZero(desc<47:40>) then
        result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
        return result;

    baseaddress = desc<39:grainsize>:Zeros(grainsize);
    if !secondstage then
        // Unpack the upper and lower table attributes
        ns_table = ns_table OR desc<63>;
        if !secondstage && !hierattrsdisabled then
            ap_table<1> = ap_table<1> OR desc<62>;       // read-only
            xn_table = xn_table OR desc<60>;
            // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
            if !singlepriv then
                pxn_table = pxn_table OR desc<59>;
                ap_table<0> = ap_table<0> OR desc<61>; // privileged

            level = level + 1;
            addrselecttop = addrselectbottom - 1;
            blocktranslate = FALSE;
            until blocktranslate;

        // Check the output address is inside the supported range
        if !IsZero(desc<47:40>) then
            result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
            return result;

        // Unpack the descriptor into address and upper and lower block attributes
        outputaddress = desc<39:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
        // Check the access flag
        if desc<10> == '0' then
            result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
            return result;

        xn = desc<54>;
        pxn = desc<53>;
        ap = desc<7:6>:'1';
        contiguousbit = desc<52>;
        nG = desc<11>;
        sh = desc<9:8>;
        memattr = desc<5:2>;
    // AttrIndx and NS bit in stage 1
    result.domain = bits(4) UNKNOWN; // Domains not used
result.level = level;
result.blocksize = 2^((3-level)*stride + grainsize);

// Stage 1 translation regimes also inherit attributes from the tables
if !secondstage then
    result.perms.xn = xn OR xn_table;
    result.perms.ap<2> = ap<2> OR ap_table<1>; // Force read-only
    // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
    if !singlepriv then
        result.perms.ap<1> = ap<1> AND NOT(ap_table<0>); // Force privileged only
        result.perms.pxn = pxn OR pxn_table;
        // Pages from Non-secure tables are marked non-global in Secure EL1&0
        if IsSecure() then
            result.nG = nG OR ns_table;
        else
            result.nG = nG;
    else
        result.perms.ap<1> = '1';
        result.perms.pxn = '0';
        result.nG = '0';
    result.GP = desc<50>; // Stage 1 block or pages might be guarded
    result.addrdesc.memattrs = AArch32.S1AttrDecode(sh, memattr<2:0>, acctype);
else
    result.perms.ap<2:1> = ap<2:1>;
    result.perms.ap<0> = '1';
    result.perms.xn = xn;
    if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
    result.perms.pxn = '0';
    result.nG = '0';
    if s2fs1walk then
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_PTW);
    else
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
    result.addrdesc.paddress.NS = '1';
    result.addrdesc.paddress.address = ZeroExtend(outputaddress);
result.addrdesc.fault = AArch32.NoFault();
result.contiguous = contiguousbit == '1';
if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;
return result;
// AArch32.TranslationTableWalkSD()
// Returns a result of a translation table walk using the Short-descriptor format
// Implementations might cache information from memory in any number of non-coherent TLB caching structures, and so avoid memory accesses that have been expressed in this pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkSD(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)
assert ELUsingAArch32(S1TranslationRegime());

// This is only called when address translation is enabled
TLBRecord result;
AddressDescriptor l1descaddr;
AddressDescriptor l2descaddr;
b bits(40) outputaddress;

// Variables for Abort functions
ipaddress = bits(40) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;
NS = bit UNKNOWN;

// Default setting of the domain
domain = bits(4) UNKNOWN;

// Determine correct Translation Table Base Register to use.
b bits(64) ttbr;
n = UInt(TTBCR.N);
if n == 0 || IsZero(vaddress<31:(32-n)>) then
ttbr = TTBR0;
disabled = (TTBCR.PD0 == '1');
else
ttbr = TTBR1;
disabled = (TTBCR.PD1 == '1');
N = 0; // TTBR1 translation always works like N=0 TTBR0 translation

// Check this Translation Table Base Register is not disabled.
if disabled then
level = 1;
result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
return result;

// Obtain descriptor from initial lookup.
l1descaddr.paddress.address = ZeroExtend(ttbr<31:14-n>:vaddress<31-n:20>:'00');
l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
IRGN = ttbr<0>:ttbr<6>; // TTBR.IRGN
RGN = ttbr<4:3>; // TTBR.RGN
SH = ttbr<1>:ttbr<5>; // TTBR.S:TTBR.NOS
l1descaddr.memattrs = WalkAttrDecode(SH, RGN, IRGN, secondstage);
if !HaveEL(EL2) || (IsSecure() && !IsSecureEL2Enabled()) then
// if only 1 stage of translation
l1descaddr2 = l1descaddr;
else
l1descaddr2 = AArch32.SecondStageWalk(l1descaddr, vaddress, acctype, iswrite, 4);
// Check for a fault on the stage 2 walk
if IsFault(l1descaddr2) then
result.addrdesc.fault = l1descaddr2.fault;
return result;

// Update virtual address for abort functions
l1descaddr2.vaddress = ZeroExtend(vaddress);
accedesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
l1desc = _Mem[l1descaddr2, 4, accedesc];
if SCTLR.EE == '1' then l1desc = BigEndianReverse(l1desc);
// Process descriptor from initial lookup.
case l1desc<1:0> of
  when '00' // Fault, Reserved
    level = 1;
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fslwalk);
    return result;
  when '01' // Large page or Small page
    domain = l1desc<8:5>;
    level = 2;
    pxn = l1desc<2>;
    NS = l1desc<3>;

    // Obtain descriptor from level 2 lookup.
l2descaddr.paddress.address = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
l2descaddr.memattrs = l1descaddr.memattrs;

    if !HaveEL(EL2) || (IsSecure() && !IsSecureEL2Enabled()) then
      // if only 1 stage of translation
      l2descaddr2 = l2descaddr;
    else
      l2descaddr2 = AArch32.SecondStageWalk(l2descaddr, vaddress, acctype, iswrite, 4);

      // Check for a fault on the stage 2 walk
      if IsFault(l2descaddr2) then
        result.addrdesc.fault = l2descaddr2.fault;
        return result;
    end;

    // Update virtual address for abort functions
    l2descaddr2.vaddress = ZeroExtend(vaddress);

    accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fslwalk, level);
l2desc = _Mem[l2descaddr2, 4, accdesc];

    if SCTLR.EE == '1' then l2desc = BigEndianReverse(l2desc);

    // Process descriptor from level 2 lookup.
if l2desc<1:0> == '00' then
  result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fslwalk);
  return result;
else
  nG = l2desc<11>;
  S = l2desc<10>;
  ap = l2desc<9,5:4>;

  if SCTLR.AFE == '1' && l2desc<4> == '0' then
    // ARMv8 VMSAv8-32 does not support hardware management of the Access flag.
    result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fslwalk);
    return result;
  end;

  if l2desc<1> == '0' then // Large page
    xn = l2desc<15>;
    tex = l2desc<14:12>;
    c = l2desc<3>;
    b = l2desc<2>;
    blocksize = 64;
    outputaddress = ZeroExtend(l2desc<31:16>:vaddress<15:0>);
  else // Small page
    tex = l2desc<8:6>;
    c = l2desc<3>;
    b = l2desc<2>;
    xn = l2desc<0>;
    blocksize = 4;
    outputaddress = ZeroExtend(l2desc<31:12>:vaddress<11:0>);
  end;
when '1x' // Section or Supersection
NS = l1desc<19>;
nG = l1desc<17>;
S = l1desc<16>;
ap = l1desc<15,11:10>;
tex = l1desc<14:12>;
xn = l1desc<4>;
c = l1desc<3>;
b = l1desc<2>;
pxn = l1desc<0>;
level = 1;

if SCTLR.AFE == '1' && l1desc<10> == '0' then
    // ARMv8 VMSAv8-32 does not support hardware management of the Access flag.
    result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;

if l1desc<18> == '0' then                          // Section
    domain = l1desc<8:5>;
    blocksize = 1024;
    outputaddress = ZeroExtend(l1desc<31:20>:vaddress<19:0>);
else                                               // Supersection
    domain = '0000';
    blocksize = 16384;
    outputaddress = l1desc<8:5>:l1desc<23:20>:l1desc<31:24>:vaddress<23:0>;

// Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes
if SCTLR.TRE == '0' then
    if RemapRegsHaveResetValues() then
        result.addrdesc.memattrs = AArch32.DefaultTEXDecode(tex, c, b, S, acctype);
    else
        result.addrdesc.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        result.addrdesc.memattrs = AArch32.RemappedTEXDecode(tex, c, b, S, acctype);

// Set the rest of the TLBRecord, try to add it to the TLB, and return it.
result.perms.ap = ap;
result.perms.xn = xn;
result.perms.pxn = pxn;
result.nG = nG;
result.domain = domain;
result.level = level;
result.blocksize = blocksize;
result.addrdesc.paddress.address = ZeroExtend(outputaddress);
result.addrdesc.paddress.NS = if IsSecure() then NS else '1';
result.addrdesc.fault = AArch32.NoFault();
return result;

Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

boolean RemapRegsHaveResetValues();
// AArch64.BreakpointMatch()
// =============================================================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccType acctype, integer size)
assert !ELUsingAArch32(S1TranslationRegime());
assert n <= UInt(ID_AArch64DFR0_EL1.BRPs);

enabled = DBGBCR_EL1[n].E == '1';
ispriv = PSTATE.EL != EL0;
linked = DBGBCR_EL1[n].BT == '0x01';
isbreakpnt = TRUE;
linked_to = FALSE;

state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC, 
linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);

value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to); 

if HaveAnyAArch32() && size == 4 then // Check second halfword
  // If the breakpoint address and BAS of an Address breakpoint match the address of the
  // second halfword of an instruction, but not the address of the first halfword, it is
  // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
  // event.
  match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
  if !value_match && match_i then
    value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
  // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
  // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
  // at the address DBGVR_EL1[n]+2.
  if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

match = value_match && state_match && enabled;
return match;
Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointValueMatch
boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n > UInt(ID_AA64DFR0_EL1.BRPs) then
  (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs), Unpredictable_BPNOTIMPL);
  assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
  if c == Constraint_DISABLED then return FALSE;
// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
// call from StateMatch for linking).
if DBGBCR_EL1[n].E == '0' then return FALSE;
context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));
// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
type = DBGBCR_EL1[n].BT;
if ((type IN {'011x','11xx'} && !HaveVirtHostExt()) ||              // Context matching
    type == '010x' ||                                             // Reserved
    type != '0x0x' && !context_aware) ||                         // Context matching
    (type == '1xxx' && !HaveEL(EL2))) then                        // EL2 extension
  (c, type) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
  assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
  if c == Constraint_DISABLED then return FALSE;
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (type == '0x0x');
match_vmid = (type == '10xx');
match_cid = (type == '001x');
match_cid1 = (type IN { '101x', 'x11x'});
match_cid2 = (type == '11xx');
linked = (type == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, of if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return FALSE;
// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return FALSE;
// Do the comparison.
if match_addr then
  byte = UInt(vaddress<1:0>);
  if HaveAnyAArch32() then
    // T32 instructions can be executed at EL0 in an AArch64 translation regime.
    assert byte IN {0,2}; // "vaddress" is halfword aligned
    byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
  else
    assert byte == 0;       // "vaddress" is word aligned
    byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is RES1
  top = AddrTop(vaddress, TRUE, PSTATE.EL);
  BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match;
else
  match_cid then
    if IsInHost() then
      BVR_match = (CONTEXTIDR_EL2 == DBGBVR_EL1[n]<31:0>);
    else
      BVR_match = (PSTATE.EL IN {EL0,EL1} && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
  else
    match_cid1 then
      BVR_match = (PSTATE.EL IN {EL0,EL1} && !IsInHost() && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
if match_vmid then
  if !Have16bitVMID() || VTCR_EL2.VS == '0' then
    vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    bvr_vmid = ZeroExtend(DBGVR_EL1[n]<39:32>, 16);
  else
    vmid = VTTBR_EL2.VMID;
    bvr_vmid = DBGVR_EL1[n]<47:32>;
  BXVR_match = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                !IsInHost() &&
    vmid == bvr_vmid);
elsif match_cid2 then
  BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                 DBGVR_EL1[n]<63:32> == CONTEXTIDR_EL2);

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);
return match;
Library pseudocode for aarch64/debug/breakpoint/AArch64.StateMatch

// AArch64.StateMatch()
// ---------------------
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
    boolean isbreakpnt, AccType acctype, boolean ispriv)
// "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

// If parameters are set to a reserved type, behaves as either disabled or a defined type
if (((HMC:SSC:PxC) IN {'011xx','100x0','101x0','11010','11101','1111x'}) ||       // Reserved
    (HMC == '0' && PxC == '00' && (!isbreakpnt || !HaveAArch32EL(EL1)))) ||    // Usr/Svc/Sys
    (SSC IN {'01','10'} && !HaveEL(EL3)) ||                                   // No EL3
    (HMC:SSC != '000' && HMC:SSC != '111' && !HaveEL(EL3) && !HaveEL(EL2))) || // No EL3/EL2
    (HMC:SSC:PxC == '11100' && !HaveEL(EL2))) then                            // No EL2
    (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);
assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
if c == Constraint_DISABLED then return FALSE;
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
EL2_match = HaveEL(EL2) && HMC == '1';
EL1_match = PxC<0> == '1';
EL0_match = PxC<1> == '1';

el = if HaveNV2Ext() && acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
if !ispriv && !isbreakpnt then
    priv_match = EL0_match;
else
    case el of
    when EL3 priv_match = EL3_match;
    when EL2 priv_match = EL2_match;
    when EL1 priv_match = EL1_match;
    when EL0 priv_match = EL0_match;

    case SSC of
    when '00' security_state_match = TRUE;         // Both
    when '01' security_state_match = !IsSecure();  // Non-secure only
    when '10' security_state_match = IsSecure();   // Secure only
    when '11' security_state_match = TRUE;         // Both

if linked then
// "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
// it must be UNPREDICTABLE whether this gives no match, or LBN is mapped to some
// UNKNOWN breakpoint that is context-aware.

lnb = Uint(LBN);
first_ctx_cmp = Uint(ID_AA64DFR0_EL1.BRPs) - Uint(ID_AA64DFR0_EL1.CTX_CMPs));
last_ctx_cmp = Uint(ID_AA64DFR0_EL1.BRPs);
if (lnb < first_ctx_cmp || lnb > last_ctx_cmp) then
    (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTX);
assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
    case c of
    when Constraint_DISABLED return FALSE;      // Disabled
    when Constraint_NONE linked = FALSE;        // No linking
    // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

if linked then
    vaddress = bits(64) UNKNOWN;
    linked_to = TRUE;
    linked_match = AArch64.BreakpointValueMatch(lnb, vaddress, linked_to);
return priv_match && security_state_match && (!linked || linked_match);
Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```java
// AArch64.GenerateDebugExceptions()
// =================================

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);
```

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```java
// AArch64.GenerateDebugExceptionsFrom()
// ====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)
    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    route_to_el2 = HaveEL(EL2) && !secure && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');
    target = (if route_to_el2 then EL2 else EL1);
    enabled = !HaveEL(EL3) || !secure || MDCR_EL3.SDD == '0';
    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from);
    return enabled;
```

Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```java
// AArch64.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()
    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1';
    for n = 0 to UInt(PMCR_EL0.N) - 1
        if HaveEL(EL2) then
            E = (if n < UInt(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
        else
            E = PMCR_EL0.E;
        if E == '1' && PMINTENSET_EL1<n> == '1' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;
    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);
    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)
    return pmuirq;
```
// AArch64.CountEvents()
// =====================
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch64.CountEvents(integer n)

assert n == 31 || n < UInt(PMCR_EL0.N);

// Event counting is disabled in Debug state
dbug = Halted();

// In Non-secure state, some counters are reserved for EL2
if HaveEL(EL2) then
    E = if n < UInt(MDCR_EL2.HPMN) || n == 31 then PMCR_EL0.E else MDCR_EL2.HPME;
else
    E = PMCR_EL0.E;

enabled = E == '1' && PMCNTENSET_EL0<n> == '1';

if !IsSecure() then
    // Event counting in Non-secure state is allowed unless all of:
    // * EL2 and the HPMD Extension are implemented
    // * Executing at EL2
    // * MDCR_EL2.HPMD == 1
    if HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(MDCR_EL2.HPMN) || n == 31) then
        prohibited = (MDCR_EL2.HPMD == '1');
    else
        prohibited = FALSE;
else
    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    prohibited = HaveEL(EL3) && MDCR_EL3.SPME == '0';

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if prohibited && !HaveSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // For the cycle counter, PMCR_EL0.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR_EL0.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M} bits
    filter = if n == 31 then PMCCFILTR else PMEVTYPER[n];

    P = filter<31>;
    U = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';
    M = if HaveEL(EL3) then filter<26> else '0';

    case PSTATE.EL of
        when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
        when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
        when EL2 filtered = (NSH == '0');
        when EL3 filtered = (M != P);
    return debug && enabled && !prohibited && !filtered;
Library pseudocode for aarch64/debug/statisticalprofiling/CheckProfilingBufferAccess

```c
// CheckProfilingBufferAccess()
// ---------------------------------------------
SysRegAccess CheckProfilingBufferAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;
    if EL2Enabled() && PSTATE.EL == EL1 && MDCR_EL2.E2PB<0> != '1' then
        return SysRegAccess_TrapToEL2;
    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;
return SysRegAccess_OK;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CheckStatisticalProfilingAccess

```c
// CheckStatisticalProfilingAccess()
// -----------------------------------------------
SysRegAccess CheckStatisticalProfilingAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;
    if EL2Enabled() && PSTATE.EL == EL1 && MDCR_EL2.TPMS == '1' then
        return SysRegAccess_TrapToEL2;
    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;
return SysRegAccess_OK;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```c
// CollectContextIDR1()
// ----------------------
boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
return PMSCR_EL1.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```c
// CollectContextIDR2()
// ----------------------
boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if EL2Enabled() then return FALSE;
return PMSCR_EL2.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```c
// CollectPhysicalAddress()
// --------------------------
boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    if (secure && HaveEL(EL2) then
        return PMSCR_EL2.PA == '1' && (el == EL2 || PMSCR_EL1.PA == '1');
else
    return PMSCR_EL1.PA == '1';
```
// CollectRecord()
// ===============

boolean CollectRecord(bits(64) events, integer total_latency, OpType optype)
assert StatisticalProfilingEnabled();
if PMSFCR_EL1.FE == '1' then
  e = events<63:48,31:24,15:12,7,5,3,1>;
  m = PMSEVFR_EL1<63:48,31:24,15:12,7,5,3,1>;
  // Check for UNPREDICTABLE case
  if IsZero(PMSEVFR_EL1) && ConstrainUnpredictableBool(Unpredictable_ZERO_PMSFR) then return FALSE;
  if !IsZero(NOT(e) AND m) then return FALSE;
if PMSFCR_EL1.FT == '1' then
  // Check for UNPREDICTABLE case
  if IsZero(PMSFCR_EL1.<B,LD,ST>) && ConstrainUnpredictableBool(Unpredictable_NO_OPS) then return FALSE;
  if IsZero(PMSFCR_EL1.B == '0' then return FALSE;
when OpType_Load if PMSFCR_EL1.LD == '0' then return FALSE;
when OpType_Store if PMSFCR_EL1.ST == '0' then return FALSE;
when OpType_LoadAtomic if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
otherwise return FALSE;
if PMSFCR_EL1.FL == '1' then
  if IsZero(PMSLATFR_EL1.MINLAT) && ConstrainUnpredictableBool(Unpredictable_ZERO_MIN_LATENCY) then // UNPREDICTABLE case
  return FALSE;
  if total_latency < UInt(PMSLATFR_EL1.MINLAT) then return FALSE;
return TRUE;

// CollectTimeStamp()
// ==================

TimeStamp CollectTimeStamp()
if !StatisticalProfilingEnabled() then return TimeStamp_None;
(sector, el) = ProfilingBufferOwner();
if el == EL2 then
  if PMSCR_EL2.TS == '0' then return TimeStamp_None;
else
  if PMSCR_EL1.TS == '0' then return TimeStamp_None;
if EL2Enabled() then
  pct = PMSCR_EL2.PCT == '1' && (el == EL2 || PMSCR_EL1.PCT == '1');
else
  pct = PMSCR_EL1.PCT == '1';
return (if pct then TimeStamp_Physical else TimeStamp_Virtual);

// ProfilingBufferEnabled()
// ========================

boolean ProfilingBufferEnabled()
if !HaveStatisticalProfiling() then return FALSE;
(sector, el) = ProfilingBufferOwner();
non_secure_bit = if secure then '0' else '1';
return (!ELUsingAArch32(el) && non_secure_bit == SCR_EL3.NS && PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
// ProfilingBufferOwner()
// ----------------------

(boolean, bits(2)) ProfilingBufferOwner()
secure = if HaveEL(EL3) then (MDCR_EL3.NSPB<1> == '0') else IsSecure();
el = if !secure && HaveEL(EL2) && MDCR_EL2.E2PB == '00' then EL2 else EL1;
return (secure, el);

// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();

// StatisticalProfilingEnabled()
// -----------------------------

boolean StatisticalProfilingEnabled()
if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
return FALSE;

in_host = EL2Enabled() && HCR_EL2.TGE == '1';
(secure, el) = ProfilingBufferOwner();
if UInt(el) < UInt(PSTATE.EL) || secure != IsSecure() || (in_host && el == EL1) then
return FALSE;

case PSTATE.EL of
when EL3 Unreachable();
when EL2 spe_bit = PMSCR_EL2.E2SPE;
when EL1 spe_bit = PMSCR_EL1.E1SPE;
when EL0 spe_bit = (if in_host then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);
return spe_bit == '1';

// SysRegAccess

Enumeration SysRegAccess { SysRegAccess_OK,
SysRegAccess_UNDEFINED,
SysRegAccess_TrapToEL1,
SysRegAccess_TrapToEL2,
SysRegAccess_TrapToEL3 };

// TimeStamp

Enumeration TimeStamp { TimeStamp_None,
TimeStamp_Virtual,
TimeStamp_Physical };

Shared Pseudocode Functions
// AArch64.TakeExceptionInDebugState()
// -----------------------------------
// Take an exception in Debug state to an Exception Level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(target_el);
    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;  PSTATE.nRW = '0';  PSTATE.SP = '1';
    SPSR[] = bits(32) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;
    if HaveSSBSExt() then PSTATE.SSBS = bits(1) UNKNOWN;
    if PSTATE.(SS,D,A,I,F) are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;
    if HaveBTIExt() then
        if exception.type == Exception_Breakpoint then
            DSPSR_EL0<11:10> = PSTATE.BTYPE;
        else
            DSPSR_EL0<11:10> = if ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE) then '00' else PSTATE.BTYPE;
        PSTATE.BTYPE = '00';
    PSTATE.IL = '0';
    if from_32 then  // Coming from AArch32
        PSTATE.IT = '00000000';  PSTATE.T = '0';  // PSTATE.J is RES0
        if HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) && SCTLR[].SPAN == '0'
            PSTATE.PAN = '1';
        if HaveMTEExt() then PSTATE.TCO = '1';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();  // Update EDSCR processor state flags.
    if sync_errors then
        SynchronizeErrors();

    EndOfInstruction();
// AArch64.WatchpointByteMatch()
// -----------------------------------------

boolean AArch64.WatchpointByteMatch(integer n, AccType acctype, bits(64) vaddress)

  el = if HaveNV2Ext() && acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
top = AddrTop(vaddress, FALSE, el);
bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;  // Word or doubleword
byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR_EL1[n].MASK);

// If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '1111111', or
// DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
// UNPREDICTABLE.
if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
  byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
else
  LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
  if !IsZero(MSB AND (MSB - 1)) then                     // Not contiguous
    byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
    bottom = 3;                                      // For the whole doubleword
// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask > 0 && mask <= 2 then
  (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
  assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
case c of
  when Constraint_DISABLED return FALSE;            // Disabled
  when ConstraintNONE mask = 0;                     // No masking
// Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value
  if mask > bottom then
    WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
    // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
    if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
      WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
      WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;
  return WVR_match && byte_select_match;
// AArch64.WatchpointMatch()  
// =========================  
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,  
   
   AccType acctype, boolean iswrite)  
   
   assert !ELUsingAArch32(S1TranslationRegime());  
   assert n <= UInt(ID_AA64DPFR0_EL1.WRPs);
   
   // "ispriv" is FALSE for LDTR/STTR instructions executed at EL1 and all  
   // load/stores at ELO, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for  
   // loads.
   enabled = DBGWCR_EL1[n].E == '1';
   linked = DBGWCR_EL1[n].WT == '1';
   isbreakpnt = FALSE;
   
   state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,  
   linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
   
   ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');
   
   value_match = FALSE;
   for byte = 0 to size - 1
   
      value_match = value_match || AArch64.WatchpointByteMatch(n, acctype, vaddress + byte);

   return value_match && state_match && ls_match && enabled;

Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

// AArch64.Abort()  
// ===============  
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

   if IsDebugException(fault) then
      if fault.acctype == AccType_IFETCH then
         if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
            AArch64.VectorCatchException(fault);
         else
            AArch64.BreakpointException(fault);
      else
         AArch64.WatchpointException(vaddress, fault);
      endelse
   else
      AArch64.InstructionAbort(vaddress, fault);
   endelse
   endelse
   AArch64.DataAbort(vaddress, fault);
// AArch64.AbortSyndrome()
// ----------------------
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception type, FaultRecord fault, bits(64) vaddress)
    exception = ExceptionSyndrome(type);
    d_side = type IN (Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint);
    exception.syndrome = AArch64.FaultSyndrome(d_side, fault);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = fault.ipaddress.NS;
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = FALSE;
    return exception;

// AArch64.CheckPCAlignment()
// -------------------------
AArch64.CheckPCAlignment()
    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();

// AArch64.DataAbort()
// ------------------
AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN (EL0,EL1) && (HCR_EL2.TGE == '1' || (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) || HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER) || IsSecondStage(fault));
    bits(64) preferred_exception_return = ThisInstrAddr();
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) && IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
    else
        exception = AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
// AArch64.InstructionAbort()
// -----------------------------------------------

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
// External aborts on instruction fetch must be taken synchronously
if HaveDoubleFaultExt() then assert fault.type != Fault.AsyncExternal;
route_to_el3 = (EL2Enabled() & PSTATE.EL IN {EL0, EL1} &
(HCR_EL2.TGE == '1' || IsSecondStage(fault)) ||
(HaveRASExt() & HCR_EL2.TEA == '1' && IsExternalAbort(fault)));

bits(64) preferred_exception_return = ThisInstrAddr();
vec_offset = 0x0;

exception = AArch64.AbortSyndrome(Exception.InstructionAbort, fault, vaddress);

if PSTATE.EL == EL3 || route_to_el3 then
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elif PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

// AArch64.PCAlignmentFault()
// -----------------------------------------------

// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

bits(64) preferred_exception_return = ThisInstrAddr();
vec_offset = 0x0;

exception = ExceptionSyndrome(Exception.PCAlignment);
exception.vaddress = ThisInstrAddr();

if UInt(PSTATE.EL) > UInt(EL1) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elif EL2Enabled() & HCR_EL2.TGE == '1' then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

// AArch64.SPAlignmentFault()
// -----------------------------------------------

// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

bits(64) preferred_exception_return = ThisInstrAddr();
vec_offset = 0x0;

exception = ExceptionSyndrome(Exception.SPAlignment);

if UInt(PSTATE.EL) > UInt(EL1) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elif EL2Enabled() & HCR_EL2.TGE == '1' then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
// BranchTargetException
// =====================
// Raise branch target exception.
AArch64.BranchTargetException(bits(52) vaddress)

route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_BranchTarget);
exception.syndrome<1:0> = PSTATE.BTYPE;
exception.syndrome<24:2> = Zeros();  // RES0

if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elseif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

// EffectiveTCF()
// ==============
// Returns the TCF field applied to Tag Check Fails in the given Exception Level

bits(2) EffectiveTCF(bits(2) el)
if el == EL3 then
tcf = SCTLR_EL3.TCF;
elif el == EL2 then
tcf = SCTLR_EL2.TCF;
elif el == EL1 then
tcf = SCTLR_EL1.TCF;
elif el == EL0 && HCR_EL2.<E2H,TGE> == '11' then
tcf = SCTLR_EL2.TCF0;
elif el == EL0 && HCR_EL2.<E2H,TGE> != '11' then
tcf = SCTLR_EL1.TCF0;
return tcf;

// RecordTagCheckFail()
// ====================
// Records a tag fail exception into the appropriate TCFR_ELx

ReportTagCheckFail(bits(2) el, bit ttbr)
if el == EL3 then
    assert ttbr == '0';
    TFSR_EL3.TF0 = '1';
elif el == EL2 then
    if ttbr == '0' then
        TFSR_EL2.TF0 = '1';
    else
        TFSR_EL2.TF1 = '1';
elif el == EL1 then
    if ttbr == '0' then
        TFSR_EL1.TF0 = '1';
    else
        TFSR_EL1.TF1 = '1';
elif el == EL0 then
    if ttbr == '0' then
        TFSR_EL0.TF0 = '1';
    else
        TFSR_EL0.TF1 = '1';
else
    TFSR_EL0.TF0 = '1';
else
    TFSR_EL0.TF1 = '1';
// TagCheckFail()
// ==============
// Handle a tag check fail condition
TagCheckFail(bits(64) vaddress, boolean iswrite)
   bits(2) tcf = EffectiveTCF(PSTATE.EL);
   if tcf == '01' then
      TagCheckFault(vaddress, iswrite);
   elsif tcf == '10' then
      ReportTagCheckFail(PSTATE.EL, vaddress<55>);

// TagCheckFault()
// ===============
// Raise a tag check fail exception.
TagCheckFault(bits(64) va, boolean write)
   bits(2) target_el;
   bits(64) preferred_exception_return = ThisInstrAddr();
   integer vect_offset = 0x0;
   if PSTATE.EL == EL0 then
      target_el = if HCR_EL2.TGE == 0 then EL1 else EL2;
   else
      target_el = PSTATE.EL;
   exception = ExceptionSyndrome(Exception_DataAbort);
   exception.syndrome<5:0> = '010001';
   if write then
      exception.syndrome<6> = '1';
   exception.vaddress = va;
   AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

// AArch64.TakePhysicalFIQException()
// ----------------------------------
AArch64.TakePhysicalFIQException()
   route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
   route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0,EL1} &&
      (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
   bits(64) preferred_exception_return = ThisInstrAddr();
   vect_offset = 0x100;
   exception = ExceptionSyndrome(Exception_FIQ);
   if route_to_el3 then
      AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
   elsif PSTATE.EL == EL2 || route_to_el2 then
      assert PSTATE.EL != EL3;
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
   else
      assert PSTATE.EL IN {EL0,EL1};
      AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
### Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
route_to_el2 = (EL2Enabled() && PSTATE.EL IN (EL0,EL1) &&
(HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;

exception = ExceptionSyndrome(Exception_IRQ);
if route_to_el3 then
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elsif PSTATE.EL == EL2 || route_to_el2 then
  assert PSTATE.EL != EL3;
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  assert PSTATE.EL IN {EL0,EL1};
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====================================

AArch64.TakePhysicalSErrorException(boolean impdef_syndrome, bits(24) syndrome)

route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
route_to_el2 = (EL2Enabled() && PSTATE.EL IN (EL0,EL1) &&
(HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x180;

exception = ExceptionSyndrome(ExceptionSError);
exception.syndrome<24> = if impdef_syndrome then '1' else '0';
exception.syndrome<23:0> = syndrome;
ClearPendingPhysicalSError();
if PSTATE.EL == EL3 || route_to_el3 then
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elsif PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

### Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =================================

AArch64.TakeVirtualFIQException()

assert EL2Enabled() && PSTATE.EL IN (EL0,EL1);
assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x100;

exception = ExceptionSyndrome(Exception_FIQ);
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```
Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

```c
// AArch64.TakeVirtualIRQException()
// -----------------------------------

AArch64.TakeVirtualIRQException()
assert EL2Enabled() && PSTATE.EL IN {EL0, EL1};
assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;

exception = ExceptionSyndrome(Exception_IRQ);
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException

```c
// AArch64.TakeVirtualSErrorException()
// -------------------------------------

AArch64.TakeVirtualSErrorException(boolean impdef_syndrome, bits(24) syndrome)
assert EL2Enabled() && PSTATE.EL IN {EL0, EL1};
assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x180;

exception = ExceptionSyndrome(ExceptionSError);
if HaveRASExt() then
    exception.syndrome<24> = VSESR_EL2.IDS;
    exception.syndrome<23:0> = VSESR_EL2.ISS;
else
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    if impdef_syndrome then exception.syndrome<23:0> = syndrome;
    ClearPendingVirtualSError();
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.BreakpointException

```c
// AArch64.BreakpointException()
// -----------------------------

AArch64.BreakpointException(FaultRecord fault)
assert PSTATE.EL != EL3;

route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

vaddress = bits(64) UNKNOWN;
exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);
if PSTATE.EL == EL2 || route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```
// AArch64.SoftwareBreakpoint()
// ----------------------------------
AArch64.SoftwareBreakpoint(bits(16) immediate)

  route_to_el2 = (EL2Enabled() && PSTATE.EL IN (EL0, EL1) &&
                  (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;

  exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
  exception.syndrome<15:0> = immediate;

  if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
  elsif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

// AArch64.SoftwareStepException()
// ---------------------------------
AAArch64.SoftwareStepException()

  assert PSTATE.EL != EL3;

  route_to_el2 = (EL2Enabled() && PSTATE.EL IN (EL0, EL1) &&
                  (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;

  exception = ExceptionSyndrome(Exception_SoftwareStep);

  if SoftwareStep_DidNotStep() then
    exception.syndrome<24> = '0';
  else
    exception.syndrome<24> = '1';
    exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';

  if PSTATE.EL == EL2 || route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

// AArch64.VectorCatchException()
// ---------------------------------
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.
AArch64.VectorCatchException(FaultRecord fault)

  assert PSTATE.EL != EL2;
  assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;

  vaddress = bits(64) UNKNOWN;

  exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
assert PSTATE.EL != EL3;

route_to_el2 = (EL2Enabled() && PSTATE.EL IN (EL0,EL1) &&
(HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);
if PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
// AArch64.ExceptionClass()
// ========================
// Return the Exception Class and Instruction Length fields for reported in ESR

(int, bit) AArch64.ExceptionClass(Exception type, bits(2) target_el)

    il = if ThisInstrLength() == 32 then '1' else '0';
    from_32 = UsingAArch32();
    assert from_32 || il == '1'; // AArch64 instructions always 32-bit

    case type of
        when Exception_Uncategorized
            ec = 0x00; il = '1';
        when Exception_WFxTrap
            ec = 0x01;
        when Exception_CP15RTTrap
            ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap
            ec = 0x04; assert from_32;
        when Exception_CP14RTTrap
            ec = 0x05; assert from_32;
        when Exception_CP14DTTrap
            ec = 0x06; assert from_32;
        when Exception_AdvSIMDFPAccessTrap
            ec = 0x07;
        when Exception_FPIDTrap
            ec = 0x08;
        when Exception_PACTrap
            ec = 0x09;
        when Exception_CP14RRTTrap
            ec = 0x0C; assert from_32;
        when Exception_BranchTarget
            ec = 0x0D;
        when Exception_IllegalState
            ec = 0x0E; il = '1';
        when Exception_SupervisorCall
            ec = 0x11;
        when Exception_HypervisorCall
            ec = 0x12;
        when Exception_MonitorCall
            ec = 0x13;
        when Exception_SystemRegisterTrap
            ec = 0x18; assert !from_32;
        when Exception_SVEAccessTrap
            ec = 0x19; assert !from_32;
        when Exception_ERetTrap
            ec = 0x1A;
        when Exception_InstructionAbort
            ec = 0x20; il = '1';
        when Exception_PCAIignment
            ec = 0x22; il = '1';
        when Exception_DataAbort
            ec = 0x24;
        when Exception_NV2DataAbort
            ec = 0x25;
        when Exception_SPAlignment
            ec = 0x26; il = '1'; assert !from_32;
        when Exception_FPTrappedException
            ec = 0x28;
        when Exception_SError
            ec = 0x2F; il = '1';
        when Exception_Breakpoint
            ec = 0x30; il = '1';
        when Exception_SoftwareStep
            ec = 0x32; il = '1';
        when Exception_Watchpoint
            ec = 0x34; il = '1';
        when Exception_SoftwareBreakpoint
            ec = 0x38;
        when Exception_VectorCatch
            ec = 0x3A; il = '1'; assert from_32;
        otherwise
            Unreachable();
    end

    if ec IN {0x20, 0x24, 0x30, 0x32, 0x34} && target_el == PSTATE.EL then
        ec = ec + 1;
    if ec IN {0x11, 0x12, 0x13, 0x28, 0x38} && !from_32 then
        ec = ec + 4;
    return (ec, il);
// AArch64.ReportException()
// ---------------------------------------
// Report syndrome information for exception taken to AArch64 state.
AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception type = exception.type;
    (ec, il) = AArch64.ExceptionClass(type, target_el);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';
    ESR[target_el] = ec<5:0>:il:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 then
        if exception.ipavalid then
            HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
            if HaveSecureEL2Ext() then
                if IsSecureEL2Enabled() then
                    HPFAR_EL2.NS = exception.NS;
                else
                    HPFAR_EL2.NS = '0';
            else
                HPFAR_EL2.NS = bits(40) UNKNOWN;

    return;

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);
AArch64.TakeReset(\(\text{boolean}\ \text{cold\_reset}\))

\begin{verbatim}
assert !\text{HighestELUsingAArch32}();

// Enter the highest implemented Exception level in AArch64 state
PSTATE.nRW = '0';
if \text{HaveEL}(\text{EL3}) then
  PSTATE.EL = \text{EL3};
elsif \text{HaveEL}(\text{EL2}) then
  PSTATE.EL = \text{EL2};
else
  PSTATE.EL = \text{EL1};

// Reset the system registers and other system components
AArch64.ResetControlRegisters(\text{cold\_reset});

// Reset all other PSTATE fields
PSTATE.SP = '1'; // Select stack pointer
PSTATE.<D,A,I,F>  = '1111'; // All asynchronous exceptions masked
PSTATE.SS = '0'; // Clear software step bit
PSTATE.DIT = '0'; // PSTATE.DIT is reset to 0 when resetting into AArch64
PSTATE.IL = '0'; // Clear Illegal Execution state bit

// All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
// below are UNKNOWN bitstrings after reset. In particular, the return information registers
// ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
// is impossible to return from a reset in an architecturally defined way.
AArch64.ResetGeneralRegisters();
AArch64.ResetSIMDFPRegisters();
AArch64.ResetSpecialRegisters();
ResetExternalDebugRegisters(\text{cold\_reset});

\begin{verbatim}
bits(64) rv; // IMPLEMENTATION DEFINED reset vector
if \text{HaveEL}(\text{EL3}) then
  rv = RVBAR_EL3;
elsif \text{HaveEL}(\text{EL2}) then
  rv = RVBAR_EL2;
else
  rv = RVBAR_EL1;
// The reset vector must be correctly aligned
assert \text{IsZero}(rv<63:\text{PAMax}()) \&\& \text{IsZero}(rv<1:0>);
BranchTo(rv, \text{BranchType\_RESET});
\end{verbatim}
\end{verbatim}
// AArch64.FPTrappedException()
// ============================
AArch64.FPTrappedException(boolean is_ase, integer element, bits(8) accumulated_exceptions)
  exception = ExceptionSyndrome(Exception_FPTrappedException);
  if is_ase then
      if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
          exception.syndrome<23> = '1'; // TFV
      else
          exception.syndrome<23> = '0'; // TFV
      else
          exception.syndrome<23> = '1'; // TFV
      exception.syndrome<10:8> = bits(3) UNKNOWN;
      if exception.syndrome<23> == '1' then
          exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
      else
          exception.syndrome<7,4:0> = bits(6) UNKNOWN;
  route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;
  if UInt(PSTATE.EL) > UInt(EL1) then
      AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
  elsif route_to_el2 then
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
      AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

// AArch64.CallHypervisor()
// ========================
// Performs a HVC call
AArch64.CallHypervisor(bits(16) immediate)
  assert HaveEL(EL2);
  if UsingAArch32() then AArch32.ITAdvance(); SSAdvance();
  bits(64) preferred_exception_return = NextInstrAddr();
  vect_offset = 0x0;
  exception = ExceptionSyndrome(Exception_HypervisorCall);
  exception.syndrome<15:0> = immediate;
  if PSTATE.EL == EL3 then
      AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
  else
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

// AArch64.CallSecureMonitor()
// --------------------------

AArch64.CallSecureMonitor(bits(16) immediate)
assert HaveEL(EL3) && !UsingAArch32(EL3);
if UsingAArch32() then AArch32.ITAdvance();
SSAdvance();
bits(64) preferred_exception_return = NextInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_MonitorCall);
exception.syndrome<15:0> = immediate;
AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

// AArch64.CallSupervisor()
// ------------------------

// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)

if UsingAArch32() then AArch32.ITAdvance();
SSAdvance();
route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
bits(64) preferred_exception_return = NextInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SupervisorCall);
exception.syndrome<15:0> = immediate;

if UInt(PSTATE.EL) > UInt(EL1) then
AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
// AArch64.TakeException()
// ================
// Take an exception to an Exception Level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception,
                                           bits(64) preferred_exception_return, integer vect_offset)

assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

sync_errors = HaveIESB() && SCTRL[].IESB == '1';
if HaveDoubleFaultExt() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
if sync_errors && InsertIESBBeforeException(target_el) then
    SynchronizeErrors();
iesb_req = FALSE;
sync_errors = FALSE;
TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
MaybeZeroSVEUppers(target_el);

if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    else
        lower_32 = ELUsingAArch32(target_el - 1);
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);
elsif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;
spsr = GetPSRFromPSTATE();
if PSTATE.EL == EL1 && target_el == EL1 && HaveNVExt() && EL2Enabled() && HCR_EL2.<NV, NV1> == '10' then
    spsr<3:2> = '10';
if HaveUAOExt() then PSTATE.UAO = '0';
if !{exception.type IN {Exception_IRQ, Exception_FIQ}} then
    AArch64.ReportException(exception, target_el);
PSTATE.EL = target_el; PSTATE.nRW = '0'; PSTATE.SP = '1';
if HaveBTIExt() then
    if { exception.type IN {Exception_SError, Exception_IRQ, Exception_FIQ, 
                            Exception_SoftwareStep, Exception_PCAlignment, Exception_ExceptionAbort, 
                            Exception_SoftwareBreakpoint, Exception_IllegalState, Exception_BranchTarget }}
        spsr_btype = PSTATE.BTYPE;
    else
        spsr_btype = if ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE) then '00' else PSTATE.BTYPE;
    spsr<11:10> = spsr_btype;

SPSR[] = spsr;
ELR[] = preferred_exception_return;
if HaveSSBSExt() then PSTATE.SSBS = SCTRL[].DSSBS;
if HaveBTIExt() then PSTATE.BTYPE = '00';
PSTATE.SS = '0';
PSTATE.<D,A,I,F> = '1111';
PSTATE.IL = '0';
if from_32 then // Coming from AArch32
PSTATE.IT = '00000000';  PSTATE.T = '0'; // PSTATE.J is RES0
if HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0)) && SCTLR[].SPAN == '1')
  PSTATE.PAN = '1';
if HaveMTEExt() then PSTATE.TCO = '1';
BranchTo (VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION);

if sync_errors then
  SynchronizeErrors();
  iesb_req = TRUE;
  TakeUnmaskedPhysicalSErrorInterrupts (iesb_req);
EndOfInstruction();

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

// AArch64.AArch32SystemAccessTrap()
// -----------------------------------
// Trapped AArch32 System register access other than due to CPTR_EL2 or CPACR_EL1.
AArch64.AArch32SystemAccessTrap(bits(2) target_el, bits(32) aarch32_instr)
  assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;
  exception = AArch64.AArch32SystemAccessTrapSyndrome(aarch32_instr);
  if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
// AArch64.AArch32 SystemAccessTrapSyndrome()
// ----------------------------------------
// Return the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS instructions, // other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32 SystemAccessTrapSyndrome(bits(32) instr)
ExceptionRecord exception;
cpnum = UInt(instr[11:8]);

bits(20) iss = Zeros();
  // MRC/MCR
  case cpnum of
    when 10 exception = ExceptionSyndrome(Exception_FPIDTrap);
    when 14 exception = ExceptionSyndrome(Exception_CP14RTTrap);
    when 15 exception = ExceptionSyndrome(Exception_CP15RTTrap);
    otherwise Unrealiable();
  endcase;
  iss[19:17] = instr[7:5]; // opc2
  iss[16:14] = instr[23:21]; // opc1
  iss[13:10] = instr[19:16]; // CRn
  if instr[20] == '1' && instr[15:12] == '1111' then // MRC, Rt==15
    iss[9:5] = '11111';
  elsif instr[20] == '0' && instr[15:12] == '1111' then // MCR, Rt==15
    iss[9:5] = bits(5) UNKNOWN;
  else
    iss[9:5] = LookUpRIndex(UInt(instr[15:12]), PSTATE.M)<4:0>;
  endcase;
  iss[4:1] = instr[3:0]; // CRm
  elsif instr[27:21] == '1100010' && instr[31:28] != '1111' then
    // MRRC/MCRR
    case cpnum of
      when 14 exception = ExceptionSyndrome(Exception_CP14RRTTrap);
      when 15 exception = ExceptionSyndrome(Exception_CP15RRTTrap);
      otherwise Unrealiable();
    endcase;
    iss[19:16] = instr[7:4]; // opc1
    if instr[19:16] == '1111' then // Rt2==15
      iss[14:10] = bits(5) UNKNOWN;
    else
      iss[14:10] = LookUpRIndex(UInt(instr[19:16]), PSTATE.M)<4:0>;
    endcase;
    if instr[15:12] == '1111' then // Rt==15
      iss[9:5] = bits(5) UNKNOWN;
    else
      iss[9:5] = LookUpRIndex(UInt(instr[15:12]), PSTATE.M)<4:0>;
    endcase;
    iss[4:1] = instr[3:0]; // CRm
      assert cpnum == 14;
      exception = ExceptionSyndrome(Exception_CP14DTTrap);
      iss[19:12] = instr[7:0]; // imm8
      iss[4] = instr[23]; // U
      if instr[19:16] == '1111' then // Rn==15, LDC(Literal addressing)/STC
        iss[9:5] = bits(5) UNKNOWN;
        iss[3] = '1';
      else
        iss[9:5] = LookUpRIndex(UInt(instr[19:16]), PSTATE.M)<4:0>;
        iss[3] = '0';
      endcase;
    else
      Unrealiable();
    endcase;
    iss[0] = instr[20]; // Direction
  endcase;
  exception.syndrome<24:20> = ConditionSyndrome();
  exception.syndrome<19:0> = iss;
endcase;
return exception;
Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

// AArch64.AdvSIMDFPAccessTrap()
// ---------------------------------------------
// Trapped access to Advanced SIMD or FP registers due to CPACR[].
AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
// AArch64.CheckAArch32SystemAccess()  
// ==============================================================  
// Check AArch32 System register access instruction for enables and disables  

AArch64.CheckAArch32SystemAccess(bits(32) instr)
    cp_num = Uint(instr<11:8>);
    assert cp_num IN {14,15};
    // Decode the AArch32 System register access instruction
    if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then  // MRC/MCR
        cp = TRUE;  cpdt = FALSE;  nreg = 1;
        opc1 = Uint(instr<23:21>);
        opc2 = Uint(instr<7:5>);
        CRn = Uint(instr<19:16>);
        CRm = Uint(instr<3:0>);
        elsif instr<31:28> != '1111' && instr<27:21> == '1100010' then  // MRRC/MCRR
            cp = TRUE;  cpdt = FALSE;  nreg = 2;
            opc1 = Uint(instr<7:4>);
            opc2 = Uint(instr<3:0>);
            CRn = Uint(instr<15:12>);
            elsif instr<31:28> != '1111' && instr<27:25> == '110' && instr<22> == '0' then  // LDC/STC
                cp = FALSE;  cpdt = TRUE;  nreg = 0;
                opc1 = 0;
                CRn = Uint(instr<15:12>);
            else
                allocated = FALSE;
            // Coarse-grain decode into CP14 or CP15 encoding space. Each of the CPxxxInstrDecode functions
            // returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
            if cp_num == 14 then
                // LDC and STC only supported for c5 in CP14 encoding space
                if cpdt && CRn != 5 then
                    allocated = FALSE;
                else
                    allocated = CP14InstrDecode(instr);
            // Coarse-grained decode of CP14 based on opc1 field
                    case opc1 of
                        when 0     allocated = CP14DebugInstrDecode(instr);
                        when 1     allocated = CP14TraceInstrDecode(instr);
                        when 7     allocated = CP14JazelleInstrDecode(instr);  // JIDR only
                        otherwise allocated = FALSE;  // All other values are unallocated
                    elseif cp_num == 15 then
                        // LDC and STC not supported in CP15 encoding space
                        if !cp then
                            allocated = FALSE;
                        else
                            allocated = CP15InstrDecode(instr);
                        // Coarse-grain traps to EL2 have a higher priority than exceptions generated because
                        // the access instruction is UNDEFINED
                        if AArch64.CheckCP15InstrCoarseTraps(CRN, nreg, CRm) then
                            // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
                            // User mode is UNDEFINED when the trap is disabled, then it is
                            // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
                            // when the trap is enabled.
                            if PSTATE.EL == EL0 && EL2Enabled() && !allocated then
                                if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at
                                EL0" then
                                    UNDEFINED;
                                    AArch64.AArch32SystemAccessTrap(EL2, instr);
                                else
                                    allocated = FALSE;
                                endif
                            else
                                allocated = FALSE;
                            endif
                        else
                            allocated = FALSE;
                        endif
                    endif
                endif
            else
                // If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
                AArch64.CheckAArch32SystemAccessTraps(instr);
            endif
        return;
```
// AArch64.CheckAArch32SystemAccessEL1Traps()
// ----------------------------------------------
// Check for configurable disables or traps to EL1 or EL2 of an AArch32 System register
// access instruction.

AArch64.CheckAArch32SystemAccessEL1Traps(instr)
assert PSTATE.EL == EL0;
trap = FALSE;

// Decode the AArch32 System register access instruction
(op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

if cp_num == 14 then
  if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 == 0) || // DBGDTRRX/DBGDTRTX
      (op == SystemAccessType_DT && CRn == 5 && opc2 == 0)) then
    trap = !Halted() && MDSCR_EL1.TDCC == '1';
  elsif opc1 == 0 then
    trap = MDSCR_EL1.TDCC == '1';
  elsif opc1 == 1 then
    trap = CPACR[].TTA == '1';
  elsif cp_num == 15 then
    if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0) || // PMCR
        (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 1) || // PMCNTEN
        (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 2) || // PMCNTENCLR
        (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 3) || // PMOVSR
        (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 5) || // PMSELR
        (op == SystemAccessType_RT && opc1 == 0 && CRn == 12 && CRm == 14 && opc2 == 0) || // PMCCNTR
        (op == SystemAccessType_RRT && opc1 == 0 && CRm == 9)) then
      trap = PMUSERENR_EL0.EN == '0' && (write || PMUSERENR_EL0.CR == '0');
    elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 0) || // PMCCNTR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8 && CRm <= 11)) then
      trap = PMUSERENR_EL0.EN == '0' && (write || PMUSERENR_EL0.ER == '0');
    elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 5 then    // PMSELR
      trap = PMUSERENR_EL0.EN == '0' && PMUSERENR_EL0.ER == '0';
    elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then // CNTP_TVAL
      trap = CNTKCTL[].EL0PTEN == '0';
    elsif op == SystemAccessType_RRT && opc1 == 1 && CRm == 14 then                             // CNTVCT
      trap = CNTKCTL[].EL0VCTEN == '0';
  else
    if trap then
      AArch64.AArch32SystemAccessTrap(EL1, instr);
```
AAs864.CheckAAs864SystemAccessEl2Traps()
// -----------------------------------------
// Check for configurable traps to El2 of an AAs864 System register access instruction.
AAs864.CheckAAs864SystemAccessEl2Traps(bits(32) instr)
assert EL2Enabled() & PSTATE.EL IN {EL0, EL1, EL2};

trap = FALSE;

// Decode the AAs864 System register access instruction
(op, cp_num, opc1, CRn, CRm, opc2, write) = AAs864.DecodeSysRegAccess(instr);

if cp_num == 14 & PSTATE.EL IN {EL0, EL1} then
  if ((op == SystemAccessType_RT & opc1 == 0 & CRn == 1 & CRm == 0 & opc2 == 0) || // DBGDRAR
      (op == SystemAccessType_RT & opc1 == 0 & CRn == 2 & CRm == 0 & opc2 == 1) || // DBGDSAR
      (op == SystemAccessType_RRT & opc1 == 0 & CRm == 2)) then // DBGDSR
    trap = MDCR_EL2.TDRA == '1' || MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1';
  elsif ((op == SystemAccessType_RT & opc1 == 0 & CRn == 1 & CRm == 0 & opc2 == 4) || // DBGOSLAR
      (op == SystemAccessType_RT & opc1 == 0 & CRn == 1 & CRm == 1 & opc2 == 4) || // DBGOSLSR
      (op == SystemAccessType_RT & opc1 == 0 & CRn == 1 & CRm == 3 & opc2 == 4) || // DBGOSDLR
      (op == SystemAccessType_RT & opc1 == 0 & CRn == 1 & CRm == 4 & opc2 == 4)) then // DBGPRCR
    trap = MDCR_EL2.TDOSA == '1' || MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1';
  elsif opc1 == 1 then
    trap = CPTR_EL2.TTA == '1';
  elsif op == SystemAccessType_RT & opc1 == 7 & CRn == 0 & CRm == 0 & opc2 == 0 then // JIDR
    trap = HCR_EL2.TID0 == '1';
  elsif cp_num == 14 & PSTATE.EL == EL2 then
    if opc1 == 1 then
      trap = CPTR_EL2.TTA == '1';
    elsif cp_num == 15 & PSTATE.EL IN {EL0, EL1} then
      if ((op == SystemAccessType_RT & opc1 == 0 & CRn == 1 & CRm == 0 & opc2 == 0) || // SCTLR
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 2 & CRm == 0 & opc2 == 0) || // TTBR0
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 2 & CRm == 0 & opc2 == 1) || // TTBR1
          (op == SystemAccessType_RT & opc1 == 1 & CRm == 2) || // TTBCR
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 3 & CRm == 0 & opc2 == 0) || // DACR
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 3 & CRm == 0 & opc2 == 1) || // DFSR
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 5 & CRm == 0 & opc2 == 0) || // DFAR
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 5 & CRm == 0 & opc2 == 1) || // IFSR
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 6 & CRm == 0 & opc2 == 0) || // DAFS
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 6 & CRm == 0 & opc2 == 1) || // IFAR
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 5 & CRm == 1 & opc2 == 0) || // ADFSR
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 5 & CRm == 1 & opc2 == 1) || // IFAR
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 10 & CRm == 2 & opc2 == 0) || // PRRR/M
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 10 & CRm == 2 & opc2 == 1) || // NMRR/M
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 10 & CRm == 3 & opc2 == 0) || // AAMIR0
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 10 & CRm == 3 & opc2 == 1) || // AAMIR1
          (op == SystemAccessType_RT & opc1 == 0 & CRn == 13 & CRm == 0 & opc2 == 1)) then // CONTEXTIDR
        trap = if write then HCR_EL2.TVM == '1' else HCR_EL2.TRVM == '1';
      elsif op == SystemAccessType_RT & opc1 == 0 & CRn == 0 then // TLBI
        trap = write & HCR_EL2.TLBI == '1';
      elsif ((op == SystemAccessType_RT & opc1 == 0 & CRn == 7 & CRm == 6 & opc2 == 2) || // DCI
         (op == SystemAccessType_RT & opc1 == 0 & CRn == 7 & CRm == 10 & opc2 == 2) || // DCI
         (op == SystemAccessType_RT & opc1 == 0 & CRn == 7 & CRm == 14 & opc2 == 2)) then // DCI
        trap = write & HCR_EL2.TSWS == '1';
      elsif ((op == SystemAccessType_RT & opc1 == 0 & CRn == 7 & CRm == 6 & opc2 == 1) || // DCI
         (op == SystemAccessType_RT & opc1 == 0 & CRn == 7 & CRm == 10 & opc2 == 1) || // DCI
         (op == SystemAccessType_RT & opc1 == 0 & CRn == 7 & CRm == 14 & opc2 == 1)) then // DCI
        trap = write & HCR_EL2.TPCP == '1';

elsif (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 1) || // ICIMVAU
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 0 && opc2 == 0) || // ICICALLU
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 1 && opc2 == 0) || // ICICALLUIS
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 11 && opc2 == 1)) then // DCCMVAU
    trap = write && HCR_EL2.TPU == '1';

elsif (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 1) || // ACTLR
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 3)) then   // ACTLR2
    trap = HCR_EL2.TACR == '1';

elsif (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 2) ||  // TCMTR
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 3) ||     // TLBTR
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 6) ||     // REVIDR
    (op == SystemAccessType_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 7)) then  // AIDR
    trap = HCR_EL2.TID1 == '1';

elsif (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 1) ||               // ID_*
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 2 && opc2 <= 7) ||     // ID_*
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm >= 3 && opc2 <= 1) ||     // Reserved
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 3 && opc2 == 2) ||     // Reserved
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 IN {4,5})) then  // Reserved
    trap = HCR_EL2.TID3 == '1';

elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0 then  // PMCR
    trap = MDCR_EL2.TPMCR == '1' || MDCR_EL2.TPM == '1';

elsif (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8) ||      // PMEVCNTR<n>/PMEVTYPER<n>
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm IN {12,13,14}) || // PM*
    (op == SystemAccessType_RRT && opc1 == 0 && CRm == 9)) then                  // PMCCNTR (MRRC/MCCR)
    trap = MDCR_EL2.TPM == '1';

elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then      // CNTP_TVAL CNTP_CTL CNTP_CVAL
    if !HaveVirtHostExt() || HCR_EL2.E2H == '0' then
        trap = CNTHCTL_EL2.EL1PCEN == '0';
    else
        trap = CNTHCTL_EL2.EL1PTEN == '0';
    end

elsif op == SystemAccessType_RRT && opc1 == 0 && CRm == 14 then                                    // CNTPCT
    trap = CNTHCTL_EL2.EL1PCTEN == '0';

elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 0) ||    // SCR
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 2) ||    // NSACR
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 12 && CRm == 0 && opc2 == 1) ||   // MVBAR
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 1) ||    // SDCR
    (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 8 && opc2 >= 4)) then // ATS12NSOxx
    trap = IsSecureEL2Enabled() && PSTATE.EL == EL1 && IsSecure() && ELUsingAArch32(EL1);

if trap then
    AArch64.AArch32SystemAccessTrap(EL2, instr);
// AArch64.CheckAArch32SystemAccessEL3Traps()  
// =============================================================================  
AArch64.CheckAArch32SystemAccessEL3Traps(bits(32) instr)  
assert HaveEL(EL3) & PSTATE.EL != EL3;  

// Decode the AArch32 System register access instruction  
(op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);  
trap = FALSE;  
if cp_num == 14 then  
if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 4 && !write) ||  
(op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 4 && write) ||  
(op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 4)) then  
trap = MDCR_EL3.TDOSA == '1';  
elsif opc1 == 0 && (!Halted() || !(op == SystemAccessType_RT && CRn == 0 && CRm == 5 && opc2 == 0)) then  
trap = MDCR_EL3.TDA == '1';  
elsif opc1 == 1 then  
trap = CPTR_EL3.TTA == '1';  
elsif cp_num == 15 then  
if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 0) ||  
(op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 2) ||  
(op == SystemAccessType_RT && opc1 == 0 && CRn == 12 && CRm == 0 && opc2 == 1) ||  
(op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 1) ||  
(op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm >= 8) then // ATS12NSOxx  
trap = PSTATE.EL == EL1 & IsSecure();  
elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8) ||  
(op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm IN {12,13,14}) ||  
(op == SystemAccessType_RT && opc1 == 0 && CRn == 9)) then  
trap = MDCR_EL3.TPM == '1';  
if trap then  
AArch64.AArch32SystemAccessTrap(EL3, instr);  

// AArch64.CheckAArch32SystemAccessTraps()  
// =============================================================================  
AArch64.CheckAArch32SystemAccessTraps(bits(32) instr)  
if PSTATE.EL == EL0 then  
AArch64.CheckAArch32SystemAccessEL1Traps(instr);  
if EL2Enabled() & PSTATE.EL IN {EL0, EL1, EL2} & !IsInHost() then  
AArch64.CheckAArch32SystemAccessEL2Traps(instr);  
AArch64.CheckAArch32SystemAccessEL3Traps(instr);  

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessEL3Traps

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessTraps
// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32 CP15 traps in HSTR_EL2 and HCR_EL2.
boolean AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
{
    // Check for coarse-grained Hyp traps
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR_EL2<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1' then
            return TRUE;

    // Check for MRC and MCR disabled by HCR_EL2.TIDCP
    if (HCR_EL2.TIDCP == '1' && nreg == 1 &&
        ((CRn == 9  && CRm IN {0,1,2, 5,6,7,8 }) ||
        (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
        return TRUE;

    return FALSE;
}

// AArch64.CheckFPAdvSIMDEnabled()
// ===============================
// Check against CPACR[]
AArch64.CheckFPAdvSIMDEnabled()
{
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        case CPACR[].FPEN of
            when 'x0'  disabled = TRUE;
            when '01'  disabled = PSTATE.EL == EL0;
            when '11'  disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);
        AArch64.CheckFPAdvSIMDTrap(); // Also check against CPTR_EL2 and CPTR_EL3
}

// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.
AArch64.CheckFPAdvSIMDTrap()
{
    if EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.TIDCP == '1' then
            case CPTR_EL2.FPEN of
                when 'x0' disabled = !(PSTATE.EL == EL1 && HCR_EL2.TGE == '1');
                when '01' disabled = (PSTATE.EL == EL0 && HCR_EL2.TGE == '1');
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
    
    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
}
// AArch64.CheckForERetTrap()
// Check for trap on ERET, ERETA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

// Non-secure EL1 execution of ERET, ERETA, ERETAB when HCR_EL2.NV bit is set, is trapped to EL2
route_to_el2 = HaveNVExt() && EL2Enabled() && PSTATE.EL == EL1 && HCR_EL2.NV == '1';
if route_to_el2 then
  ExceptionRecord exception;
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;
  exception = ExceptionSyndrome(Exception_ERetTrap);
  if !eret_with_pac then // ERET
    exception.syndrome<1> = '0';
    exception.syndrome<0> = '0'; // RES0
  else
    exception.syndrome<1> = '1';
    if pac_uses_key_a then // ERETA
      exception.syndrome<0> = '0';
    else // ERETAB
      exception.syndrome<0> = '1';
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

// AArch64.CheckForSMCUndefOrTrap()
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)

route_to_el2 = EL2Enabled() && PSTATE.EL == EL1 && HCR_EL2.TSC == '1';
if PSTATE.EL == EL0 then UNDEFINED;
if !HaveEL(EL3) then
  if EL2Enabled() && PSTATE.EL == EL1 then
    if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
      route_to_el2 = TRUE;
    else
      UNDEFINED;
  else
    UNDEFINED;
else
  route_to_el2 = EL2Enabled() && PSTATE.EL == EL1 && HCR_EL2.TSC == '1';
if route_to_el2 then
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;
  exception = ExceptionSyndrome(Exception_MonitorCall);
  exception.syndrome<15:0> = imm;
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

// AArch64.CheckForWFxTrap()
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)

assert HaveEL(target_el);

case target_el of
  when EL1 trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
  when EL2 trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
  when EL3 trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
if trap then
  AArch64.WFxTrap(target_el, is_wfe);
// AArch64.CheckIllegalState()
// ------------------------------------------
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.
AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_IllegalState);
        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elseif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.TakeException

// AArch64.MonitorModeTrap()
// --------------------------
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode
AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = ExceptionSyndrome(Exception_Uncategorized);
    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemRegisterTrap

// AArch64.SystemRegisterTrap()
// -----------------------------
// Trapped system register access other than due to CPTR_EL2 and CPACR_EL1
AArch64.SystemRegisterTrap(bits(2) target_el, bits(2) op0, bits(3) op2, bits(3) op1, bits(4) crn,
    bits(5) rt, bits(4) crm, bit dir)
    assert UInt(target_el) >= UInt(PSTATE.EL);
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
    exception.syndrome<21:20> = op0;
    exception.syndrome<19:17> = op2;
    exception.syndrome<16:14> = op1;
    exception.syndrome<13:10> = crn;
    exception.syndrome<9:5> = rt;
    exception.syndrome<4:1> = crm;
    exception.syndrome<0> = dir;
    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/traps/AArch64.UndefinedFault

// AArch64.UndefinedFault()
// ------------------------

AArch64.UndefinedFault()
    route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

// AArch64.WFxTrap()
// ------------------

AArch64.WFxTrap(bits(2) target_el, boolean is_wfe)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<0> = if is_wfe then '1' else '0';

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

// CheckFPAdvSIMDEnabled64()
// ------------------------
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
// AArch64.CreateFaultRecord
// ----------------------------------------

FaultRecord AArch64.CreateFaultRecord(Fault type, bits(52) ipaddress, bit NS, Integer level, AccType acctype, boolean write, bit extflag, bits(2) errortype, boolean secondstage, boolean s2fs1walk)

FaultRecord fault;
    fault.type = type;
    fault.domain = bits(4) UNKNOWN; // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN; // Not used from AArch64
    fault.errortype = errortype;
    fault.ipaddress.NS = NS;
    fault.ipaddress.address = ipaddress;
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;

    return fault;

Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

// AArch64.FaultSyndrome()
// ------------------------

// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to an Exception Level using AArch64.

bits(25) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(25) iss = Zeros();
    if HaveRASExt() && IsExternalSyncAbort(fault) then iss<12:11> = fault.errortype; // SET
    if d_side then
        if IsSecondStage(fault) && fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
            iss<13> = '1'; // Value of '1' indicates fault is generated by use of VNCR_EL2
        if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC, AccType_AT} then
            iss<8> = '1'; iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fs1walk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.type, fault.level);
    else
        iss<6> = if fault.write then '1' else '0';
    return iss;
// AArch64.ExclusiveMonitorsPass()
// -----------------------------------------------

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    if passed then
        ClearExclusiveLocal(ProcessorID());
    if memaddrdesc.memattrs.shareable then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    return passed;

Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);

Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA

// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.
AArch64.SetExclusiveMonitors(bits(64) address, integer size)

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  return;
if memaddrdesc.memattrs.shareable then
  MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
AArch64.MarkExclusiveVA(address, ProcessorID(), size);

// Fully fused multiply-add and halve
result_value = (3.0 + (value1 * value2)) / 2.0;
if result_value == 0.0 then
  // Sign of exact zero result depends on rounding mode
  sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
  result = FPZero(sign);
else
  result = FPRound(result_value, FPCR);
return result;
Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```c
// FPRecipStepFused()
// ---------------

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
            endif
        endif
    else
        // Fully fused multiply-add
        result_value = 2.0 + (value1 * value2);
        if result_value == 0.0 then
            // Sign of exact zero result depends on rounding mode
            sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
            result = FPZero(sign);
        else
            result = FPRound(result_value, FPCR);
        endif
    endif
    return result;
```

Library pseudocode for aarch64/functions/memory/AArch64.CheckAlignment

```c
// AArch64.CheckAlignment()
// -----------------------

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype, boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic = acctype IN {AccType_ATOMIC, AccType_ATOMICCRN, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICCRN, AccType_ORDEREDATOMICCRNCRN, AccType_ORDEREDATOMICCRNCRNCRN, AccType_ORDEREDATOMICCRNCRNCRNCRN};
    ordered = acctype IN {AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_LIMITEDORDEREDRW, AccType_LIMITEDORDEREDRWCRN, AccType_LIMITEDORDEREDRWCRNCRN, AccType_LIMITEDORDEREDRWCRNCRNCRN, AccType_LIMITEDORDEREDRWCRNCRNCRNCRN};
    vector = acctype == AccType_VEC;
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveUA16Ext () then
        check = (UInt(address<0:+4>) + alignment > 16) && ((ordered && SCTLR[].nAA == '0') || atomic);
    else check = atomic || ordered;
    endif
    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
    endif
    return aligned;
```
// AArch64.MemSingle[] - non-assignment (read) form
// ==============================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned]
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
bits(size*8) value;
iswrite = FALSE;

// MMU or MPU
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
  if AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
    bits(4) ptag = TransformTag(ZeroExtend(address, 64));
    if !CheckTag(memaddrdesc, ptag, iswrite) then
      TagCheckFail(ZeroExtend(address, 64), iswrite);
  value = _Mem[memaddrdesc, size, accdesc];
return value;

// AArch64.MemSingle[] - assignment (write) form
// ==============================================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) value
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;

// MMU or MPU
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareable then
  ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
  if AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
    bits(4) ptag = TransformTag(ZeroExtend(address, 64));
    if !CheckTag(memaddrdesc, ptag, iswrite) then
      TagCheckFail(ZeroExtend(address, 64), iswrite);
  _Mem[memaddrdesc, size, accdesc] = value;
return;
// AddressWithAllocationTag()
// -------------------------
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as "0000000".
bits(64) AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag = allocation_tag - ('000':address<55>);
    result<59:56> = tag;
    return result;

// AllocationTagFromAddress()
// --------------------------
// Generate a Tag from a 64-bit value containing a Logical Address Tag.
// If access to Allocation Tags is disabled, this function returns "0000000".
bits(4) AllocationTagFromAddress(bits(64) tagged_address)
    bits(4) logical_tag = tagged_address<59:56>;
    bits(4) tag = logical_tag + ('000':tagged_address<55>);
    return tag;

// CheckSPAlignment()
// ---------------------
// Check correct stack pointer alignment for AArch64 state.
CheckSPAlignment()
    bits(64) sp = SP[];
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR[].SA0 != '0');
    else
        stack_align_check = (SCTLR[].SA != '0');
        if stack_align_check && sp != Align(sp, 16) then
            AArch64.SPAlignmentFault();
    return;

// CheckTag()
// ----------
// Performs a Tag Check operation for a memory access and returns
// whether the check passed
boolean CheckTag(AddressDescriptor memaddrdesc, bits(4) ptag, boolean write)
    if memaddrdesc.memattrs.tagged then
        bits(64) paddress = ZeroExtend(memaddrdesc.paddress.address);
        return ptag == MemTag[paddress];
    else
        return TRUE;

// IsBlockDescriptorNTBitValid()
// If the implementation supports changing the block size without a break-before-make
// approach, then for implementations that have level 1 or 2 support, the nT bit in
// the block descriptor is valid.
boolean IsBlockDescriptorNTBitValid();
// Mem[] - non-assignment (read) form
// ----------------------------------------------
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
  assert size IN {1, 2, 4, 8, 16};
  bits(size*8) value;
  boolean iswrite = FALSE;

  aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
  if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
    atomic = aligned;
  else
    // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
    // 64-bit aligned.
    atomic = address == Align(address, 8);

  if !atomic then
    assert size > 1;
    value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];
    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
    // access will generate an Alignment Fault, as to get this far means the first byte did
    // not, so we must be changing to a new translation page.
    if !aligned then
      c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
      assert c IN {Constraint_FAULT, Constraint_NONE};
      if c == Constraint_NONE then aligned = TRUE;

      for i = 1 to size-1
        value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
      elseif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        value<63:0> = AArch64.MemSingle[address, 8, acctype, aligned];
        value<127:64> = AArch64.MemSingle[address+8, 8, acctype, aligned];
      else
        value = AArch64.MemSingle[address, size, acctype, aligned];

      if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '1') || BigEndian() then
        value = BigEndianReverse(value);
    return value;

  return value;

// Mem[] - assignment (write) form
// ----------------------------------------------
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
  boolean iswrite = TRUE;

  if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '1') || BigEndian() then
    value = BigEndianReverse(value);

  aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
  if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
    atomic = aligned;
  else
    // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
    // 64-bit aligned.
    atomic = address == Align(address, 8);

  if !atomic then
    assert size > 1;
    AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;
    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
    // access will generate an Alignment Fault, as to get this far means the first byte did
    // not, so we must be changing to a new translation page.
    if !aligned then
      c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
      assert c IN {Constraint_FAULT, Constraint_NONE};
if c == Constraint_NONE then aligned = TRUE;

for i = 1 to size-1
    AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>
elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
    AArch64.MemSingle[address, 8, acctype, aligned] = value<63:0>
    AArch64.MemSingle[address+8, 8, acctype, aligned] = value<127:64>
else
    AArch64.MemSingle[address, size, acctype, aligned] = value;
return;

Library pseudocode for aarch64/functions/memory/MemTag

// MemTag[] - non-assignment (read) form
// --------------------------------------
// Load an Allocation Tag from memory.

bits(4) MemTag[bits(64) address] = AddressDescriptor memaddrdesc;
    bits(4) value;
    iswrite = FALSE;

memaddrdesc = AArch64.TranslateAddress(address, AccType_NORMAL, iswrite, TRUE, TAG_GRANULE);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
// Return the granule tag if tagging is enabled...
if AllocationTagAccessIsEnabled() then
    return _MemTag[memaddrdesc];
else
    // ...otherwise read tag as zero.
    return '0000';

// MemTag[] - assignment (write) form
// ----------------------------------
// Store an Allocation Tag to memory.

MemTag[bits(64) address] = bits(4) value
    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

// Stores of allocation tags must be aligned
if address != Align(address, TAG_GRANULE) then
    boolean secondstage = FALSE;
    AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, iswrite, secondstage));
wasaligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, AccType_NORMAL, iswrite, wasaligned, TAG_GRANULE);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
// Memory array access
if AllocationTagAccessIsEnabled() then
    _MemTag[memaddrdesc] = value;

Library pseudocode for aarch64/functions/memory/TransformTag

// TransformTag()
// ---------------
// Apply tag transformation rules.

bits(4) TransformTag(bits(64) vaddr) =
    bits(4) vtag = vaddr<59:56>;
    bits(4) tagdelta = ZeroExtend(vaddr<55>);
    bits(4) ptag = vtag + tagdelta;
    return ptag;
// boolean AccessIsTagChecked()
// -----------------------------------
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AccessIsTagChecked(bits(64) vaddr, AccType acctype)
  if PSTATE.M<4> == '1' then return FALSE;
    if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
      return FALSE;
    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') then
      return FALSE;
    if !AllocationTagAccessIsEnabled() then
      return FALSE;
    if acctype IN {AccType_IFETCH, AccType_PTW} then
      return FALSE;
    if acctype == AccType_NV2REGISTER then
      return FALSE;
    if PSTATE.TCO=='1' then
      return FALSE;
    if IsNonTagCheckedInstruction() then
      return FALSE;
  return TRUE;
Library pseudocode for aarch64/functions/pac/addpac/AddPAC

// AddPAC()
// =========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
bits(64) PAC;
bits(64) result;
bits(64) ext_ptr;
bits(64) extfield;
bit selbit;
boolean tbi = CalculateTBI(ptr, data);
integer top_bit = if tbi then 55 else 63;

// If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
// the pointer to select between upper and lower ranges, and preserve this.
// This handles the awkward case where there is apparently no correct choice between
// the upper and lower address range - i.e. an addr of 1xxxxxxxx... with TB10=0 and TB11=1
// and 0xxxxxxxx1 with TB11=0 and TB10=1:
if TPtrHasUpperAndLowerAddRanges() then
  assert S1TranslationRegime() IN {EL1, EL2};
  if S1TranslationRegime() == EL1 then
    // EL1 translation regime registers
    if data then
      selbit = if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then ptr<55> else ptr<63>;
    else
      selbit = ptr<55>;
  else
    // EL2 translation regime registers
    if data then
      selbit = if (HaveEL(EL2) && TCR_EL2.TBI1 == '1') ||
                (HaveEL(EL2) && TCR_EL2.TBID1 == '0') then ptr<55> else ptr<63>;
    else
      selbit = if tbi then ptr<55> else ptr<63>;
  integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

  // The pointer authentication code field takes all the available bits in between
  extfield = Replicate(selbit, 64);

  // Compute the pointer authentication code for a ptr with good extension bits
  if tbi then
    ext_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
  else
    ext_ptr = extfield<64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

  PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

  // Check if the ptr has good extension bits and corrupt the pointer authentication code if not;
  if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
    PAC<top_bit-1> = NOT(PAC<top_bit-1>);

  // Preserve the determination between upper and lower address at bit<55> and insert PAC
  if tbi then
    result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
  else
    result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
return result;
// AddPACDA()
// ===========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APDAKey_EL1;

APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;

case PSTATE.EL of
    when EL0
        boolean IsEL1Regime = S1TranslationRegime() == EL1;
        Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
        TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                   (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
        TrapEL3 = HaveEL(EL3) && SCR_EL3/API == '0';
    when EL1
        Enable = SCTLR_EL1.EnDA;
        TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        TrapEL3 = HaveEL(EL3) && SCR_EL3/API == '0';
    when EL2
        Enable = SCTLR_EL2.EnDA;
        TrapEL2 = FALSE;
        TrapEL3 = HaveEL(EL3) && SCR_EL3/API == '0';
    when EL3
        Enable = SCTLR_EL3.EnDA;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APDAKey_EL1, TRUE);
 Library pseudocode for aarch64/functions/pac/addpacdb/AddPACDB

```c
// AddPACDB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APDBKey_EL1;

APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

case PSTATE.EL of
    when EL0
        boolean IsEL1Regime = S1TranslationRegime() == EL1;
        Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
        TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                   (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
        Enable = SCTLR_EL1.EnDB;
        TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
        Enable = SCTLR_EL2.EnDB;
        TrapEL2 = FALSE;
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
        Enable = SCTLR_EL3.EnDB;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APDBKey_EL1, TRUE);
```

Shared Pseudocode Functions
Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

// AddPACGA()
// =========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of X, Y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(128) APGAKey_EL1;

APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;

Case PSTATE.EL of
when EL0
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
when EL1
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
when EL2
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
when EL3
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;
if TrapEL2 then TrapPACUse(EL2);
elseif TrapEL3 then TrapPACUse(EL3);
else return ComputePAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0><63:32>:Zeros(32));
// AddPACIA()
// ===========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;

case PSTATE.EL of
    when EL0
        boolean IsEL1Regime = S1TranslationRegime() == EL1;
        Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
        TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
            (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
        Enable = SCTLR_EL1.EnIA;
        TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
        Enable = SCTLR_EL2.EnIA;
        TrapEL2 = FALSE;
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
        Enable = SCTLR_EL3.EnIA;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APIAKey_EL1, FALSE);
// AddPACIB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APIBKey_EL1;

APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;

case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnIB;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnIB;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnIB;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

if Enable == '0' then return X;
elseif TrapEL2 then TrapPACUse(EL2);
elseif TrapEL3 then TrapPACUse(EL3);
else return AddPAC(X, Y, APIBKey_EL1, FALSE);
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit keynumber)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = CalculateTBI(ptr, data);
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);
    if tbi then
        original_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<64-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
            result = original_ptr;
        else
            error_code = keynumber:NOT(keynumber);
            result = original_ptr<63:55>:error_code:original_ptr<52:0>;
    else
        if ((PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit>) &&
            (PAC<63:56> == ptr<63:56>)) then
            result = original_ptr;
        else
            error_code = keynumber:NOT(keynumber);
            result = original_ptr<63>:error_code:original_ptr<60:0>;
    return result;
Library pseudocode for aarch64/functions/pac/authda/AuthDA

// AuthDA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA()!

bits(64) AuthDA(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APDAKey_EL1;

APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;

case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
               (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnDA;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnDA;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnDA;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

if Enable == '0' then return X;
elsif TrapEL2 then TrapPACUse(EL2);
elsif TrapEL3 then TrapPACUse(EL3);
else return Auth(X, Y, APDAKey_EL1, TRUE, '0');
Library pseudocode for aarch64/functions/pac/authdb/AuthDB

// AuthDB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APDBKey_EL1;

  APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
      (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnDB;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnDB;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnDB;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else return Auth(X, Y, APDBKey_EL1, TRUE, '1');
library pseudocode for aarch64/functions/pac/authia/AuthIA

// AuthIA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) X, bits(64) Y)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APIAKey_EL1;

  APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;

  case PSTATE.EL of
    when EL0
      boolean IsEL1Regime = S1TranslationRegime() == EL1;
      Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
      TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                  (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
      Enable = SCTLR_EL1.EnIA;
      TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
      Enable = SCTLR_EL2.EnIA;
      TrapEL2 = FALSE;
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
      Enable = SCTLR_EL3.EnIA;
      TrapEL2 = FALSE;
      TrapEL3 = FALSE;
  end case

  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else return Auth(X, Y, APIAKey_EL1, FALSE, '0');
/** AuthIB() */

// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code field bits with the extension of the address bits. The instruction checks a pointer authentication code in the pointer authentication code field bits of X, using the same algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) X, bits(64) Y)
  boolean TrapEL2;
  boolean TrapEl3;
  bits(1) Enable;
  bits(128) APIBKey_EL1;

  APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;

  case PSTATE.El of
    when EL0
      boolean IsEL1Regime = SlTranslationRegime() == EL1;
      Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
      TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                  (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEl3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
      Enable = SCTLR_EL1.EnIB;
      TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
      TrapEl3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
      Enable = SCTLR_EL2.EnIB;
      TrapEL2 = FALSE;
      TrapEl3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
      Enable = SCTLR_EL3.EnIB;
      TrapEL2 = FALSE;
      TrapEl3 = FALSE;
  end;

  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEl3 then TrapPACUse(EL3);
  else return Auth(X, Y, APIBKey_EL1, FALSE, '1');
// CalculateBottomPACBit()
// =======================

integer CalculateBottomPACBit(bit top_bit)
integer tsz_field;

if PtrHasUpperAndLowerAddRanges() then
    assert S1TranslationRegime() IN {EL1, EL2};
    if S1TranslationRegime() == EL1 then
        // EL1 translation regime registers
        tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
        using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
    else
        // EL2 translation regime registers
        assert HaveEL(EL2);
        tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
        using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
    else
        tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else UInt(TCR_EL3.T0SZ);
        using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';

    max_limit_tsz_field = (if !HaveSmallPageTblExt() then 39 else if using64k then 47 else 48);

    if tsz_field > max_limit_tsz_field then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        tszmin = if using64k && VAMax() == 52 then 12 else 16;
        if tsz_field < tszmin then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_NONE};
            if c == Constraint_FORCE then tsz_field = tszmin;

        return (64-tsz_field);
    return (64-tsz_field);
return (64-tsz_field);
// CalculateTBI()
// ==============

boolean CalculateTBI(bits(64) ptr, boolean data)
boolean tbi = FALSE;

if _PtrHasUpperAndLowerAddRanges_() then
    assert _S1TranslationRegime_() IN {EL1, EL2};
    if _S1TranslationRegime_() == EL1 then
        // EL1 translation regime registers
        if data then
            tbi = if ptr<55> == '1' then TCR_EL1.TBI1 == '1' else TCR_EL1.TBI0 == '1';
        else
            if ptr<55> == '1' then
                tbi = TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0';
            else
                tbi = TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0';
        else
            // EL2 translation regime registers
            if data then
                tbi = if ptr<55> == '1' then TCR_EL2.TBI1 == '1' else TCR_EL2.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0';
                else
                    tbi = TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0';
    elsif PSTATE.EL == EL2 then
        tbi = if data then TCR_EL2.TBI=='1' else TCR_EL2.TBI=='1' && TCR_EL2.TBID=='0';
    elsif PSTATE.EL == EL3 then
        tbi = if data then TCR_EL2.TBI=='1' else TCR_EL2.TBI=='1' && TCR_EL2.TBID=='0';
else
    tbi = if data then TCR_EL3.TBI=='1' else TCR_EL3.TBI=='1' && TCR_EL3.TBID=='0';

return tbi;
array bits(64) RC[0..4];

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
  bits(64) workingval;
  bits(64) runningmod;
  bits(64) roundkey;
  bits(64) modk0;
  constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;
  RC[0] = 0x0000000000000000<63:0>;
  RC[1] = 0x13198A2E03707344<63:0>;
  RC[2] = 0xA4093822299F31D0<63:0>;
  RC[3] = 0x082EFA98E4E6C89<63:0>;
  RC[4] = 0x452821E638D01377<63:0>;

  modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
  runningmod = modifier;
  workingval = data EOR key0;
  for i = 0 to 4
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = workingval EOR RC[i];
    if i > 0 then
      workingval = PACCellShuffle(workingval);
      workingval = PACMult(workingval);
      workingval = PACSub(workingval);
    runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = PACSub(workingval);
  workingval = PACCellShuffle(workingval);
  workingval = PACMult(workingval);
  workingval = PACSub(workingval);
  workingval = PACCellShuffle(workingval);
  workingval = PACMult(workingval);
  workingval = PACSub(workingval);
  workingval = PACCellShuffle(workingval);
  for i = 0 to 4
    workingval = PACInvSub(workingval);
    if i < 4 then
      workingval = PACMult(workingval);
      runningmod = TweakInvShuffle(runningmod<63:0>);
      roundkey = key1 EOR runningmod;
      workingval = workingval EOR RC[4-i];
    workingval = workingval EOR roundkey;
    workingval = workingval EOR Alpha;
  workingval = workingval EOR modk0;
  return workingval;
Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```cpp
// PACCellInvShuffle()
// ===============
bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```cpp
// PACCellShuffle()
// ================
bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<43:40>;
    outdata<59:56> = indata<63:60>;
    return outdata;
```
Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```c
// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher

    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '0101';
            when '0001'  Toutput<4*i+3:4*i> = '1110';
            when '0010'  Toutput<4*i+3:4*i> = '1101';
            when '0011'  Toutput<4*i+3:4*i> = '1000';
            when '0100'  Toutput<4*i+3:4*i> = '1010';
            when '0101'  Toutput<4*i+3:4*i> = '1011';
            when '0110'  Toutput<4*i+3:4*i> = '0001';
            when '0111'  Toutput<4*i+3:4*i> = '1001';
            when '1000'  Toutput<4*i+3:4*i> = '0110';
            when '1001'  Toutput<4*i+3:4*i> = '1111';
            when '1010'  Toutput<4*i+3:4*i> = '0000';
            when '1011'  Toutput<4*i+3:4*i> = '0100';
            when '1100'  Toutput<4*i+3:4*i> = '1100';
            when '1101'  Toutput<4*i+3:4*i> = '0111';
            when '1110'  Toutput<4*i+3:4*i> = '0011';
        return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACMult

```c
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)

    bits(64) Soutput;
    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);

    Soutput<4*i+3:4*i> = t3<3:0>;
    Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
    Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
    Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;

    return Soutput;
```
Library pseudocode for aarch64/functions/pac/computepac/PACSub

```plaintext
// PACSub()
// =========

bits(64) PACSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000'  Toutput<4*i+3:4*i> = '1011';
        when '0001'  Toutput<4*i+3:4*i> = '0110';
        when '0010'  Toutput<4*i+3:4*i> = '1000';
        when '0011'  Toutput<4*i+3:4*i> = '1111';
        when '0100'  Toutput<4*i+3:4*i> = '1100';
        when '0101'  Toutput<4*i+3:4*i> = '0000';
        when '0110'  Toutput<4*i+3:4*i> = '1001';
        when '0111'  Toutput<4*i+3:4*i> = '1110';
        when '1000'  Toutput<4*i+3:4*i> = '0011';
        when '1001'  Toutput<4*i+3:4*i> = '0111';
        when '1010'  Toutput<4*i+3:4*i> = '0100';
        when '1011'  Toutput<4*i+3:4*i> = '0101';
        when '1100'  Toutput<4*i+3:4*i> = '0010';
        when '1101'  Toutput<4*i+3:4*i> = '0011';
        when '1110'  Toutput<4*i+3:4*i> = '1010';
        when '1111'  Toutput<4*i+3:4*i> = '1010';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/RotCell

```plaintext
// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```plaintext
// TweakCellInvRot()
// ================

bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```plaintext
// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
return outcell;
```
Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

// TweakInvShuffle()
// -------------------

bits(64) TweakInvShuffle(bits(64) indata)
bits(64) outdata;
outdata<3:0> = TweakCellInvRot(indata<51:48>);
outdata<7:4> = indata<55:52>;
outdata<11:8> = indata<23:20>;
outdata<15:12> = indata<27:24>;
outdata<19:16> = indata<3:0>;
outdata<23:20> = indata<7:4>;
outdata<27:24> = TweakCellInvRot(indata<11:8>);
outdata<31:28> = indata<15:12>;
outdata<35:32> = TweakCellInvRot(indata<31:28>);
outdata<39:36> = TweakCellInvRot(indata<63:60>);
outdata<43:40> = TweakCellInvRot(indata<59:56>);
outdata<47:44> = TweakCellInvRot(indata<19:16>);
outdata<51:48> = indata<35:32>;
outdata<55:52> = indata<39:36>;
outdata<59:56> = indata<43:40>;
outdata<63:60> = TweakCellInvRot(indata<47:44>);
return outdata;

Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

// TweakShuffle()
// ---------------

bits(64) TweakShuffle(bits(64) indata)
bits(64) outdata;
outdata<3:0> = indata<19:16>;
outdata<7:4> = indata<23:20>;
outdata<11:8> = TweakCellRot(indata<27:24>);
outdata<15:12> = indata<31:28>;
outdata<19:16> = TweakCellRot(indata<47:44>);
outdata<23:20> = indata<11:8>;
outdata<27:24> = indata<15:12>;
outdata<31:28> = TweakCellRot(indata<35:32>);
outdata<35:32> = TweakCellRot(indata<51:48>);
outdata<39:36> = indata<55:52>;
outdata<43:40> = indata<59:56>;
outdata<47:44> = TweakCellRot(indata<63:60>);
outdata<51:48> = TweakCellRot(indata<3:0>);
outdata<55:52> = indata<7:4>;
outdata<59:56> = TweakCellRot(indata<43:40>);
outdata<63:60> = TweakCellRot(indata<39:36>);
return outdata;

Library pseudocode for aarch64/functions/pac/pac/HavePACExt

// HavePACExt()
// -------------

boolean HavePACExt()
return HasArchVersion(ARMv8p3);

Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

// PtrHasUpperAndLowerAddRanges()
// ----------------------------------

// Returns TRUE if the pointer has upper and lower address ranges

boolean PtrHasUpperAndLowerAddRanges()
return PSTATE.EL == EL1 || PSTATE.EL == EL0 || (PSTATE.EL == EL2 && HCR_EL2.E2H == '1');
Library pseudocode for aarch64/functions/pac/strip/Strip

```c
// Strip()
// ========
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
code field bits with the extension of the address bits. This can apply to either
instructions or data, where, as the use of tagged pointers is distinct, it might be
handled differently.

bits(64) Strip(bits(64) A, boolean data)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(64) original_ptr;
  bits(64) extfield;
  boolean tbi = CalculateTBI(A, data);
  integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
  extfield = Replicate(A<55>, 64);
  if tbi then
    original_ptr = A<63:56>:extfield< 56-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
  else
    original_ptr = extfield< 64-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
  case PSTATE.EL of
    when EL0
      TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                  (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
      TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
      TrapEL2 = FALSE;
      TrapEL3 = False;
    when EL3
      TrapEL2 = False;
      TrapEL3 = FALSE;
  if TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else return original_ptr;
```

Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```c
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
  assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
  bits(64) preferred_exception_return = ThisInstrAddr();
  ExceptionRecord exception;
  vect_offset = 0;
  exception = ExceptionSyndrome(Exception_PACTrap);
  AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```
AArch64.ESBOperation()

// Perform the AArch64 ESBOperation, either for ESBOperation executed in AArch64 state, or for ESBOperation in AArch32 state when SError interrupts are routed to an Exception level using AArch64

AArch64.ESBOperation()

route_to_el3 = (HaveEL(EL3) && SCR_EL3.EA == '1');
route_to_el2 = (EL2Enabled() &&
(HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

target = (if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1);

if target == EL1 then
    mask_active = (PSTATE.EL IN {EL0, EL1});
elsesif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
    mask_active = (PSTATE.EL IN {EL0, EL2});
else
    mask_active = (PSTATE.EL == target);

mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));
intdis = (Halted() || ExternalDebugInterruptsDisabled(target));
masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending
if IsPhysicalSErrorPending() && masked then
    // This function might be called for an interworking case, and INTdis is masking the SError interrupt.
    if ELUsingAArch32(S1TranslationRegime()) then
        syndrome32 = AArch32.PhysicalSErrorSyndrome();
        DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
    else
        implicit_esb = FALSE;
        syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
        DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
        ClearPendingPhysicalSError();               // Set ISR_EL1.A to 0
    return;

Library pseudocode for aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);

Library pseudocode for aarch64/functions/ras/AArch64.ReportDeferredSError

// AArch64.ReportDeferredSError()
// ---------------------------------------
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31> = '1';       // A
    target<24> = syndrome<24>; // IDS
    target<23:0> = syndrome<23:0>; // ISS
    return target;
Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation

```assembly
// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
assert EL2Enabled() && PSTATE.EL IN {EL0, EL1};

// If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
// SError interrupt might be pending
vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
vmasked = vintdis || PSTATE.A == '1';

// Check for a masked virtual SError pending
if vSEI_pending && vmasked then
  // This function might be called for the interworking case, and INTdis is masking
  // the virtual SError interrupt.
  if ELUsingAArch32(EL1) then
    VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
  else
    VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
    HCR_EL2.VSE = '0';  // Clear pending virtual SError
  return;

Library pseudocode for aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```assembly
// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
assert UsingAArch32(); // Always called from AArch32 state before entering AArch64 state

if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
  first = 0; last = 14; include_R15 = FALSE;
elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
  first = 0; last = 30; include_R15 = FALSE;
else
  first = 0; last = 30; include_R15 = TRUE;

for n = first to last
  if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
    _R[n]<63:32> = Zeros();

return;

Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```assembly
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()
for i = 0 to 30
  X[i] = bits(64) UNKNOWN;

return;
```
```plaintext
// AArch64.ResetSIMDFPRegisters()
// ==============================
AArch64.ResetSIMDFPRegisters()
    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;
    return;

// AArch64.ResetSpecialRegisters()
// ===============================
AArch64.ResetSpecialRegisters()
    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(32) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(32) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;
    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAAArch32EL(EL1) then
        SPSR_fiq = bits(32) UNKNOWN;
        SPSR_irq = bits(32) UNKNOWN;
        SPSR_abt = bits(32) UNKNOWN;
        SPSR_und = bits(32) UNKNOWN;
    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;
    return;

AArch64.ResetSystemRegisters(boolean cold_reset);

// PC - non-assignment form
// ========================
// Read program counter.

bits(64) PC[]
    return _PC;
```
// SP[] - assignment form
// =======================
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
assert width IN {32,64};
if PSTATE.SP == '0' then
    SP_EL0 = ZeroExtend(value);
else
    case PSTATE.EL of
        when EL0 SP_EL0 = ZeroExtend(value);
        when EL1 SP_EL1 = ZeroExtend(value);
        when EL2 SP_EL2 = ZeroExtend(value);
        when EL3 SP_EL3 = ZeroExtend(value);
    return;

// SP[] - non-assignment form
// ===========================
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
assert width IN {8,16,32,64};
if PSTATE.SP == '0' then
    return SP_EL0<width-1:0>;
else
    case PSTATE.EL of
        when EL0 return SP_EL0<width-1:0>;
        when EL1 return SP_EL1<width-1:0>;
        when EL2 return SP_EL2<width-1:0>;
        when EL3 return SP_EL3<width-1:0>;
    return;

Library pseudocode for aarch64/functions/registers/V

// V[] - assignment form
// ======================

V[integer n] = bits(width) value
assert n >= 0 && n <= 31;
assert width IN {8,16,32,64,128};
integer vlen = if IsSVEEnabled(PSTATE.EL) then VL else 128;
if ConstrainsUnpredictableBool(Unpredictable_SVEZEROUPPER) then
    _Z[n] = ZeroExtend(value);
else
    _Z[n]<vlen-1:0> = ZeroExtend(value);

// V[] - non-assignment form
// =========================

bits(width) V[integer n]
assert n >= 0 && n <= 31;
assert width IN {8,16,32,64,128};
return _Z[n]<width-1:0>;
Library pseudocode for aarch64/functions/registers/Vpart

// Vpart[] - non-assignment form
// ----------------------------------------

bits(width) Vpart(integer n, integer part)
assert n >= 0 && n <= 31;
assert part IN {0, 1};
if part == 0 then
    assert width IN {8,16,32,64};
    return V[n];
else
    assert width == 64;
    return _V[n]<(width * 2)-1:width>;

// Vpart[] - assignment form
// ----------------------------------------

Vpart(integer n, integer part) = bits(width) value
assert n >= 0 && n <= 31;
assert part IN {0, 1};
if part == 0 then
    assert width IN {8,16,32,64};
    V[n] = value;
else
    assert width == 64;
    bits(64) vreg = V[n];
    V[n] = value63:0 : vreg;

Library pseudocode for aarch64/functions/registers/X

// X[] - assignment form
// ------------------------
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X(integer n) = bits(width) value
assert n >= 0 && n <= 31;
assert width IN {32,64};
if n != 31 then
    _R[n] = ZeroExtend(value);
return;

// X[] - non-assignment form
// ------------------------
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X(integer n)
assert n >= 0 && n <= 31;
assert width IN {8,16,32,64};
if n != 31 then
    return _R[n]<width-1:0>;
else
    return Zeros(width);
// AArch32.IsFPEnabled()
// ---------------

boolean AArch32.IsFPEnabled(bits(2) el)
if el == EL0 && !ELUsingAArch32(EL1) then
  return AArch64.IsFPEnabled(el);
if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() then
  // Check if access disabled in NSACR
  if NSACR.cp10 == '0' then return FALSE;
if el IN {EL0, EL1} then
  // Check if access disabled in CPACR
  case CPACR.cp10 of
    when 'x0' disabled = TRUE;
    when '01' disabled = (el == EL0);
    when '11' disabled = FALSE;
  if disabled then return FALSE;
if el IN {EL0, EL1, EL2} then
  if EL2Enabled() then
    if !ELUsingAArch32(EL2) then
      if CPTR_EL2.TFP == '1' then return FALSE;
    else
      if HCPTR.TCP10 == '1' then return FALSE;
if HaveEL(EL3) && ELUsingAArch32(EL3) then
  // Check if access disabled in CPTR_EL3
  if CPTR_EL3.TFP == '1' then return FALSE;
return TRUE;

// AArch64.IsFPEnabled()
// ---------------

boolean AArch64.IsFPEnabled(bits(2) el)
// Check if access disabled in CPACR_EL1
if el IN {EL0, EL1} then
  // Check FP&SIMD at EL0/EL1
  case CPACR[].FPEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = (el == EL0);
    when '11' disabled = FALSE;
  if disabled then return FALSE;
// Check if access disabled in CPTR_EL2
if el IN {EL0, EL1, EL2} && EL2Enabled() then
  if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
    if CPTR_EL2.FPEN == 'x0' then return FALSE;
  else
    if CPTR_EL2.TFP == '1' then return FALSE;
// Check if access disabled in CPTR_EL3
if HaveEL(EL3) then
  if CPTR_EL3.TFP == '1' then return FALSE;
return TRUE;
// CeilPow2()
// ============
// For a positive integer X, return the smallest power of 2 \geq X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;

// CheckSVEEnabled()
// ===============

CheckSVEEnabled()
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} then
        // Check SVE at EL0/EL1
        case CPACR[].ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL1);
        // Check FP&SIMD at EL0/EL1
        case CPACR[].FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            if CPTR_EL2.ZEN == 'x0' then SVEAccessTrap(EL2);
            if CPTR_EL2.FPEN == 'x0' then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then SVEAccessTrap(EL3);
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
// DecodePredCount()
// ================

integer DecodePredCount(bits(5) pattern, integer esize)
    integer elements = VL DIV esize;
    integer numElem;
    case pattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise numElem = 0;
    return numElem;

// ElemFFR[] - non-assignment form
// ===============================

bit ElemFFR[integer e, integer esize]
    return ElemP[FFR, e, esize];

// ElemFFR[] - assignment form
// ===========================

ElemFFR[integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= PL;
    _FFR<n+psize-1:n> = ZeroExtend(value, psize);
    return;

// ElemP[] - non-assignment form
// =============================

bit ElemP[bits(N) pred, integer e, integer esize]
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;

// ElemP[] - assignment form
// =========================

ElemP[bits(N) &pred, integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= N;
    pred<n+psize-1:n> = ZeroExtend(value, psize);
    return;
Library pseudocode for aarch64/functions/sve/FFR

// FFR[] - non-assignment form
// ---------------------------------

bits(width) FFR[]
assert width == PL;
return _FFR<width-1:0>;

// FFR[] - assignment form
// ---------------------------------

FFR[] = bits(width) value
assert width == PL;
if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
  _FFR = ZeroExtend(value);
else
  _FFR<width-1:0> = value;

Library pseudocode for aarch64/functions/sve/FPCompareNE

// FPCompareNE()  
// ===============

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
  result = TRUE;
  if type1==FPType_SNaN || type2==FPType_SNaN then
    FPProcessException(FPExc_InvalidOp, fpcr);
  else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 != value2);
  return result;

Library pseudocode for aarch64/functions/sve/FPCompareUN

// FPCompareUN()
// ===============

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
if type1==FPType_SNaN || type2==FPType_SNaN then
  FPProcessException(FPExc_InvalidOp, fpcr);
return (type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN);

Library pseudocode for aarch64/functions/sve/FPConvertSVE

// FPConvertSVE()
// ===============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr, FPRounding rounding)
fpcr.AHP = '0';
return FPConvert(op, fpcr, rounding);

// FPConvertSVE()
// ===============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr)
fpcr.AHP = '0';
return FPConvert(op, fpcr, FPRoundingMode(fpcr));
// FPExpA()
// ========

bits(N) FPExpA(bits(N) op, FPCRType fpcr)
assert N IN {16,32,64};
bits(N) result;
bits(N) coeff;
integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
coeff = FPExpCoefficient[idx];
if N == 16 then
    result<15:0> = '0':op<9:5>:coeff<9:0>;
elsif N == 32 then
    result<31:0> = '0':op<13:6>:coeff<22:0>;
else // N == 64
    result<63:0> = '0':op<16:6>:coeff<51:0>;
return result;
// FPExpCoefficient()
// ---------------------

bits(N) FPExpCoefficient[integer index]
assert N IN {16,32,64};
integer result;

if N == 16 then
    case index of
    when 0 result = 0x0000;
    when 1 result = 0x0016;
    when 2 result = 0x002d;
    when 3 result = 0x0045;
    when 4 result = 0x005d;
    when 5 result = 0x0075;
    when 6 result = 0x008e;
    when 7 result = 0x00a8;
    when 8 result = 0x00c2;
    when 9 result = 0x00dc;
    when 10 result = 0x0114;
    when 11 result = 0x0114;
    when 12 result = 0x0130;
    when 13 result = 0x014d;
    when 14 result = 0x016b;
    when 15 result = 0x0189;
    when 16 result = 0x01a8;
    when 17 result = 0x01c8;
    when 18 result = 0x01e8;
    when 19 result = 0x0209;
    when 20 result = 0x022b;
    when 21 result = 0x024e;
    when 22 result = 0x0271;
    when 23 result = 0x0295;
    when 24 result = 0x02ba;
    when 25 result = 0x02e0;
    when 26 result = 0x0306;
    when 27 result = 0x032e;
    when 28 result = 0x0356;
    when 29 result = 0x037f;
    when 30 result = 0x03a9;
    when 31 result = 0x03d4;
elseif N == 32 then
    case index of
    when 0 result = 0x000000;
    when 1 result = 0x0164d2;
    when 2 result = 0x02cd87;
    when 3 result = 0x043a29;
    when 4 result = 0x05aac3;
    when 5 result = 0x071f62;
    when 6 result = 0x08980f;
    when 7 result = 0x0a14d5;
    when 8 result = 0x0b95c2;
    when 9 result = 0x0d1ad6;
    when 10 result = 0x0ea43a;
    when 11 result = 0x1031dc;
    when 12 result = 0x11c3d3;
    when 13 result = 0x135a2b;
    when 14 result = 0x14f4f0;
    when 15 result = 0x16942d;
    when 16 result = 0x1837f0;
    when 17 result = 0x19e046;
    when 18 result = 0x1b8d3a;
    when 19 result = 0x1d3eda;
    when 20 result = 0x1ef532;
    when 21 result = 0x20b051;
    when 22 result = 0x227043;
    when 23 result = 0x243516;
    when 24 result = 0x25fed7;
    when 25 result = 0x27cd94;

Shared Pseudocode Functions
Page 1856
when 26 result = 0x29a15b;
when 27 result = 0x2b7a3a;
when 28 result = 0x2d583f;
when 29 result = 0x2f3b79;
when 30 result = 0x3111c4;
when 31 result = 0x3311c4;
when 32 result = 0x3504f3;
when 33 result = 0x36fd92;
when 34 result = 0x38fbaf;
when 35 result = 0x3aff5b;
when 36 result = 0x3d08a4;
when 37 result = 0x3f179a;
when 38 result = 0x412c4d;
when 39 result = 0x4346cd;
when 40 result = 0x45672a;
when 41 result = 0x478d75;
when 42 result = 0x49b9be;
when 43 result = 0x4bec15;
when 44 result = 0x4e248c;
when 45 result = 0x506334;
when 46 result = 0x52a81e;
when 47 result = 0x54f35b;
when 48 result = 0x5744fd;
when 49 result = 0x599d1f;
when 50 result = 0x5bfbb8;
when 51 result = 0x5e60f5;
when 52 result = 0x60ccdf;
when 53 result = 0x633f89;
when 54 result = 0x65b907;
when 55 result = 0x68396a;
when 56 result = 0x6ac0c7;
when 57 result = 0x6d4f30;
when 58 result = 0x6fe4ba;
when 59 result = 0x728177;
when 60 result = 0x75257d;
when 61 result = 0x77d0df;
when 62 result = 0x7a83b3;
when 63 result = 0x7d3e0c;
else // N == 64
  case index of
    when 0 result = 0x0000000000000000;
    when 1 result = 0x02c9a3e778061;
    when 2 result = 0x059b0d3158574;
    when 3 result = 0x0874518759bc8;
    when 4 result = 0x0b5586cf9890f;
    when 5 result = 0x0ec32d31a2;
    when 6 result = 0x11301d0125b51;
    when 7 result = 0x1429aaa92de0;
    when 8 result = 0x172b83c7d517b;
    when 9 result = 0x1a35beb6fcb75;
    when 10 result = 0x1dd873168b9a;
    when 11 result = 0x2063b88628cd6;
    when 12 result = 0x2387a6e756238;
    when 13 result = 0x26b456e27cdd;
    when 14 result = 0x29e9df51fde1;
    when 15 result = 0x2d285a6e4030b;
    when 16 result = 0x306fe0a31b715;
    when 17 result = 0x33c06b26416ff;
    when 18 result = 0x371a7373aa9cb;
    when 19 result = 0x3a7db34859ff7;
    when 20 result = 0x3dea64123422;
    when 21 result = 0x4160a21f72e2a;
    when 22 result = 0x44e086061892d;
    when 23 result = 0x486a2b5c13cdo;
    when 24 result = 0x4bfdd5362a27;
    when 25 result = 0x4f9b2769d2ca7;
    when 26 result = 0x5342b569d4f82;
    when 27 result = 0x56f4736b527da;
    when 28 result = 0x5ab070dd485429;
    Shared Pseudocode Functions
when 29 result = 0x5E76F15AD2148;
when 30 result = 0x6247EB03A5585;
when 31 result = 0x6623882552225;
when 32 result = 0x6A09E667F3BCD;
when 33 result = 0x6DFB23C651A2F;
when 34 result = 0x71F75E8EC5F74;
when 35 result = 0x75FEB564267C9;
when 36 result = 0x7A11473EB0187;
when 37 result = 0x7E2F336CF4E62;
when 38 result = 0x82589994CCE13;
when 39 result = 0x868D99B4492ED;
when 40 result = 0x8ACE5422A0DB;
when 41 result = 0x8F1AE99157736;
when 42 result = 0x93737B0CDCE5;
when 43 result = 0x97D829FDE4E50;
when 44 result = 0xC49182A3F090;
when 45 result = 0xA0C6675DE565;
when 46 result = 0xA5503B2355D;
when 47 result = 0xA9E665579FDBF;
when 48 result = 0xAE89F995AD3AD;
when 49 result = 0xB33A2B84F15FB;
when 50 result = 0xB7F6F285E47;
when 51 result = 0xBC1E904BC1D2;
when 52 result = 0xC199BD85529C;
when 53 result = 0xC67F12E57D14B;
when 54 result = 0xCB720DCEFB9069;
when 55 result = 0xD072D4A0789C7;
when 56 result = 0xD5818DCFBA487;
when 57 result = 0xDA9E603DB285;
when 58 result = 0xDFC97337B85B;
when 59 result = 0xE502EE78B3FF6;
when 60 result = 0xEAC4FA5D690DA;
when 61 result = 0xEF01E615A27;
when 62 result = 0xF5075B58E540;
when 63 result = 0xFA7C1819E90D8;

return result<N-1:0>;

library pseudocode for aarch64/functions/sve/FPMinNormal

// FPMinNormal()
// =============

bits(N) FPMinNormal(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = Zeros(E-1):'1';
frac = Zeros(F);
return sign : exp : frac;

library pseudocode for aarch64/functions/sve/FPOne

// FPOne()
// =======

bits(N) FPOne(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0':Ones(E-1);
frac = Zeros(F);
return sign : exp : frac;
Library pseudocode for aarch64/functions/sve/FPPointFive

```plaintext
// FPPointFive()
// =============
bits(N) FPPointFive(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0': ones(E-2):'0';
frac = zeros(F);
return sign : exp : frac;
```

Library pseudocode for aarch64/functions/sve/FPProcess

```plaintext
// FPProcess()
// ===========
bits(N) FPProcess(bits(N) input)
bits(N) result;
assert N IN {16,32,64};
(type, sign, value) = FPUnpack(input, FPCR);
if type == FPType_SNaN || type == FPType_QNaN then
  result = FPProcessNaN(type, input, FPCR);
elsif type == FPType_Infinity then
  result = FPInfinity(sign);
elsif type == FPType_Zero then
  result = FPZero(sign);
else
  result = FPRound(value, FPCR);
return result;
```

Library pseudocode for aarch64/functions/sve/FPScale

```plaintext
// FPScale()
// =========
bits(N) FPScale(bits (N) op, integer scale, FPCRType fpcr)
assert N IN {16,32,64};
(type, sign, value) = FPUnpack(op, fpcr);
if type == FPType_SNaN || type == FPType_QNaN then
  result = FPProcessNaN(type, op, fpcr);
elsif type == FPType_Zero then
  result = FPZero(sign);
elsif type == FPType_Infinity then
  result = FPInfinity(sign);
else
  result = FPRound(value * (2.0^scale), fpcr);
return result;
```
// FPTrigMAdd()
// ============

bits(N) FPTrigMAdd(integer x, bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16, 32, 64};
assert x >= 0;
assert x < 8;
bits(N) coeff;

if op2<N-1> == '1' then
    x = x + 8;
    op2<N-1> = '0';

coeff = FPTrigMAddCoefficient[x];
result = FPMulAdd(coeff, op1, op2, fpcr);

return result;
// FPTrigMAddCoefficient()
// -----------------------------------

bits(N) FPTrigMAddCoefficient[integer index]
assert N IN {16, 32, 64};
integer result;

if N == 16 then
case index of
  when 0 result = 0x3c00;
  when 1 result = 0xb155;
  when 2 result = 0x2030;
  when 3 result = 0x0000;
  when 4 result = 0x0000;
  when 5 result = 0x0000;
  when 6 result = 0x0000;
  when 7 result = 0x0000;
  when 8 result = 0x3c00;
  when 9 result = 0xb800;
  when 10 result = 0x293a;
  when 11 result = 0x0000;
  when 12 result = 0x0000;
  when 13 result = 0x0000;
  when 14 result = 0x0000;
  when 15 result = 0x0000;
elsif N == 32 then
case index of
  when 0 result = 0x3f800000;
  when 1 result = 0xbe2aaaab;
  when 2 result = 0x3c088886;
  when 3 result = 0xb95008b9;
  when 4 result = 0x36369d6d;
  when 5 result = 0x00000000;
  when 6 result = 0x00000000;
  when 7 result = 0x00000000;
  when 8 result = 0x3f800000;
  when 9 result = 0xbf000000;
  when 10 result = 0x3d2aaaa6;
  when 11 result = 0xbab60705;
  when 12 result = 0x37cd37cc;
  when 13 result = 0x00000000;
  when 14 result = 0x00000000;
  when 15 result = 0x00000000;
else // N == 64
 case index of
  when 0 result = 0x3ff0000000000000;
  when 1 result = 0xbfc555555555543;
  when 2 result = 0x3f81111111110f30c;
  when 3 result = 0xbf2a01a019b92fc6;
  when 4 result = 0x3ec71de351f3d22b;
  when 5 result = 0xbe5ae5e2b60f7b91;
  when 6 result = 0x3de5d8408868552f;
  when 7 result = 0x0000000000000000;
  when 8 result = 0x3ff0000000000000;
  when 9 result = 0xbfe0000000000000;
  when 10 result = 0x3fa555555555536;
  when 11 result = 0xbf56c16c16c13a0b;
  when 12 result = 0x3efa01a019b1e8d8;
  when 13 result = 0xbe927e4f7282f468;
  when 14 result = 0x3e21ee96d2641b13;
  when 15 result = 0xbda8f76380fbb401;

return result<N-1:0>;
Library pseudocode for aarch64/functions/sve/FPTrigSMul

```c
// FPTrigSMul()
// ============

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
result = FPMul(op1, op1, fpcr);
(type, sign, value) = FPUnpack(result, fpcr);
if (type != FPType_QNaN) && (type != FPType_SNaN) then
  result<N-1> = op2<0>;
return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigSSel

```c
// FPTrigSSel()
// ============

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
assert N IN {16,32,64};
bits(N) result;
if op2<0> == '1' then
  result = FPOne(op2<1>);
else
  result = op1;
  result<N-1> = result<N-1> EOR op2<1>;
return result;
```

Library pseudocode for aarch64/functions/sve/FirstActive

```c
// FirstActive()
// =============

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
integer elements = N DIV (esize DIV 8);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
return '0';
```

Library pseudocode for aarch64/functions/sve/FloorPow2

```c
// FloorPow2()
// ===========

// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
assert x >= 0;
integer n = 1;
if x == 0 then return 0;
while x >= 2^n do
  n = n + 1;
return 2^(n - 1);
```

Library pseudocode for aarch64/functions/sve/HaveSVE

```c
// HaveSVE()
// =========

boolean HaveSVE()
return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Have SVE ISA";
```
// ImplementedSVEVectorLength()
// -----------------------------
// Reduce SVE vector length to a supported value (e.g. power of two)

integer ImplementedSVEVectorLength(integer nbits)
return integer IMPLEMENTATION_DEFINED;

// IsEven()
// -------

boolean IsEven(integer val)
return val MOD 2 == 0;

// IsFPEnabled()
// =============

boolean IsFPEnabled(bits(2) el)
if ELUsingAArch32(el) then
  return AArch32.IsFPEnabled(el);
else
  return AArch64.IsFPEnabled(el);

// IsSVEEnabled()
// ==============

boolean IsSVEEnabled(bits(2) el)
if ELUsingAArch32(el) then
  return FALSE;

// Check if access disabled in CPACR_EL1
if el IN {EL0, EL1} then
  // Check SVE at EL0/EL1
  case CPACR[].ZEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = (el == EL0);
    when '11' disabled = FALSE;
  if disabled then return FALSE;

// Check if access disabled in CPTR_EL2
if el IN {EL0, EL1, EL2} && EL2Enabled() then
  if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
    if CPTR_EL2.ZEN == 'x0' then return FALSE;
  else
    if CPTR_EL2.TZ == '1' then return FALSE;

// Check if access disabled in CPTR_EL3
if HaveEL(EL3) then
  if CPTR_EL3.EZ == '0' then return FALSE;

return TRUE;
Library pseudocode for aarch64/functions/sve/LastActive

```plaintext
// LastActive()
// ============

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

Library pseudocode for aarch64/functions/sve/LastActiveElement

```plaintext
// LastActiveElement()
// ===============

integer LastActiveElement(bits(N) mask, integer esize)
    assert esize IN {8, 16, 32, 64};
    integer elements = VL DIV esize;
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return e;
    return -1;
```

Library pseudocode for aarch64/functions/sve/MAX_PL

```plaintext
constant integer MAX_PL = 256;
```

Library pseudocode for aarch64/functions/sve/MAX_VL

```plaintext
constant integer MAX_VL = 2048;
```

Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

```plaintext
// MaybeZeroSVEUppers()
// ================

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;
    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then return;
    if target_el == EL3 then
        if EL2Enabled() then lower_enabled = IsFPEnabled(EL2);
        else lower_enabled = IsFPEnabled(EL1);
    else lower_enabled = IsFPEnabled(target_el - 1);
    if lower_enabled then
        integer vl = if IsSVEEnabled(PSTATE.EL) then VL else 128;
        integer pl = vl DIV 8;
        for n = 0 to 31
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _Z[n] = ZeroExtend(_Z[n]<vl-1:0>);
            for n = 0 to 15
                if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                    _P[n] = ZeroExtend(_P[n]<pl-1:0>);
                if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                    _FFR = ZeroExtend(_FFR<pl-1:0>);
```

Shared Pseudocode Functions
// MemNF[] - non-assignment form
// ------------------------------------------
(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccType acctype]
assert size IN {1, 2, 4, 8, 16};
bits(8*size) value;
aligned = (address == Align(address, size));
A = SCTLR[].A;
if !aligned && (A == '1') then
  return (bits(8*size) UNKNOWN, TRUE);
atomic = aligned || size == 1;
if !atomic then
  (value<7:0>, bad) = MemSingleNF[address, 1, acctype, aligned];
  if bad then
    return (bits(8*size) UNKNOWN, TRUE);
  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  if !aligned then
    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE;
  for i = 1 to size-1
    (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, acctype, aligned];
    if bad then
      return (bits(8*size) UNKNOWN, TRUE);
  else
    (value, bad) = MemSingleNF[address, size, acctype, aligned];
    if bad then
      return (bits(8*size) UNKNOWN, TRUE);
  if BigEndian() then
    value = BigEndianReverse(value);
return (value, FALSE);
// MemSingleNF[] - non-assignment form
// ==========================

(bits(8*size), boolean) MemSingleNF(bits(64) address, integer size, AccType acctype, boolean wasaligned, bits(8*size) value;
boolean iswrite = FALSE;
AddressDescriptor memaddrdesc;

// Implementation may suppress NF load for any reason
if ConstrainUnpredictableBool(Unpredictable_NONFAULT) then
  return (bits(8*size) UNKNOWN, TRUE);

// MMU or MPU
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

// Non-fault load from Device memory must not be performed externally
if memaddrdesc.memattrs.type == MemType_Device then
  return (bits(8*size) UNKNOWN, TRUE);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  return (bits(8*size) UNKNOWN, TRUE);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
  if AccessIsTagChecked(address, acctype) then
    bits(4) ptag = TransformTag(address);
    if !CheckTag(memaddrdesc, ptag, iswrite) then
      return (bits(8*size) UNKNOWN, TRUE);
  value = _Mem[memaddrdesc, size, accdesc];
return (value, FALSE);

// NoneActive() // ============

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
integer elements = N DIV (esize DIV 8);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' && ElemP[x, e, esize] == '1' then return '0';
return '1';

// P[] - non-assignment form
// ==========================

bits(width) P[integer n]
assert n >= 0 && n <= 31;
assert width == PL;
return _P[n]<width-1:0>;

// P[] - assignment form
// =====================

P[integer n] = bits(width) value
assert n >= 0 && n <= 31;
assert width == PL;
if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
  _P[n] = ZeroExtend(value);
else
  _P[n]<width-1:0> = value;
Library pseudocode for aarch64/functions/sve/PL

// PL - non-assignment form
// -------------------------

integer PL
  return VL DIV 8;

Library pseudocode for aarch64/functions/sve/PredTest

// PredTest()
// =========

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
  bit n = FirstActive(mask, result, esize);
  bit z = NoneActive(mask, result, esize);
  bit c = NOT LastActive(mask, result, esize);
  bit v = '0';
  return n:z:c:v;

Library pseudocode for aarch64/functions/sve/ReducePredicated

// ReducePredicated()
// ==================

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
  assert(N == M * 8);
  integer p2bits = CeilPow2(N);
  bits(p2bits) operand;
  integer elements = p2bits DIV esize;
  for e = 0 to elements-1
    if e * esize < N && ElemP[mask, e, esize] == '1' then
      Elem[operand, e, esize] = Elem[input, e, esize];
    else
      Elem[operand, e, esize] = identity;
  return Reduce(op, operand, esize);

Library pseudocode for aarch64/functions/sve/Reverse

// Reverse()
// =========

// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
  bits(N) result;
  integer sw = N DIV M;
  assert N == sw * M;
  for s = 0 to sw-1
    Elem[result, sw - 1 - s, M] = Elem[word, s, M];
  return result;
Library pseudocode for aarch64/functions/sve/SVEAccessTrap

```c
// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.
SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) & target_el != EL0 & HaveEL(target_el);  
    route_to_el2 = target_el == EL1 & EL2Enabled() & HCR_EL2.TGE == '1';
    exception = ExceptionSyndrome(Exception_SVEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    if route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/sve/SVECmp

```c
enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

```c
// SVEMoveMaskPreferred()
// ================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.
boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, _) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE); 
    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffffxy' or '00000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then return FALSE;
        // Check for 'ffxyfffffffffy' or '00xy0000000000xy'
        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> & & imm<31:16> == imm<15:0> & & (imm<15:8> == imm<7:0>) then return FALSE;
    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffffxy00' or '000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then return FALSE;
        // Check for 'ffxyy00ffxy00' or '00xy000000xy00'
        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> & & imm<31:16> == imm<15:0> then return FALSE;
    return TRUE;
```
Library pseudocode for aarch64/functions/sve/System

array bits(MAX_VL) _Z[0..31];
array bits(MAX_PL) _P[0..15];
bits(MAX_PL) _FFR;

Library pseudocode for aarch64/functions/sve/VL

// VL - non-assignment form
// -----------------------

integer VL
    integer vl;
    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(ZCR_EL1.LEN);
    endif
    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(ZCR_EL2.LEN);
    endlif
    elsif EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
        vl = Min(vl, UInt(ZCR_EL2.LEN));
    endelsif
    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    endif
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));
    endelsif
    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);
    return vl;

Library pseudocode for aarch64/functions/sve/Z

// Z[] - non-assignment form
// -------------------------

bits(width) Z[integer n]
    assert n >= 0 && n <= 31;
    assert width == VL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// ----------------------

Z[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == VL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value);
    else
        _Z[n]<width-1:0> = value;

Library pseudocode for aarch64/functions/sysregisters/CNTKCTL

// CNTKCTL[] - non-assignment form
// -------------------------------

CNTKCTLType CNTKCTL[]
    bits(32) r;
    if IsInHost() then
        r = CNTHCTL_EL2;
    return r;
    r = CNTHCTL_EL1;
    return r;
Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

type CNTKCTLType;

Library pseudocode for aarch64/functions/sysregisters/CPACR

// CPACR[] - non-assignment form
// --------------------------------------------
CPACRType CPACR[]
    bits(32) r;
    if IsInHost() then
        r = CPTR_EL2;
        return r;
    r = CPACR_EL1;
    return r;

Library pseudocode for aarch64/functions/sysregisters/CPACRType

type CPACRType;

Library pseudocode for aarch64/functions/sysregisters/ELR

// ELR[] - non-assignment form
// ---------------------------
bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1 r = ELR_EL1;
        when EL2 r = ELR_EL2;
        when EL3 r = ELR_EL3;
        otherwise unreachable();
    return r;

// ELR[] - non-assignment form
// ---------------------------
bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// ------------------------
ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1 ELR_EL1 = r;
        when EL2 ELR_EL2 = r;
        when EL3 ELR_EL3 = r;
        otherwise unreachable();
    return;

// ELR[] - assignment form
// ------------------------
ELR[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR[PSTATE.EL] = value;
    return;
Library pseudocode for aarch64/functions/sysregisters/ESR

```c
// ESR[] - non-assignment form
// ---------------------------

ESRType ESR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise unreachable();
    return r;

// ESR[] - non-assignment form
// ---------------------------

ESRType ESR[]
    return ESR[SiTranslationRegime()];

// ESR[] - assignment form
// -----------------------

ESR[bits(2) regime] = ESRType value
    bits(32) r = value;
    case regime of
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise unreachable();
    return;

// ESR[] - assignment form
// -----------------------

ESR[] = ESRType value
    ESR[SiTranslationRegime()] = value;
```

Library pseudocode for aarch64/functions/sysregisters/ESRType

```c
type ESRType;
```
Library pseudocode for aarch64/functions/sysregisters/FAR

```plaintext
// FAR[] - non-assignment form
// -----------------------------------------------

bits(64) FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        when EL3  r = FAR_EL3;
        otherwise unreachable();
    return r;

// FAR[] - non-assignment form
// -----------------------------------------------

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// ------------------------

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        when EL3  FAR_EL3 = r;
        otherwise unreachable();
    return;

// FAR[] - assignment form
// ------------------------

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/MAIR

```plaintext
// MAIR[] - non-assignment form
// -----------------------------------------------

MAIRType MAIR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = MAIR_EL1;
        when EL2  r = MAIR_EL2;
        when EL3  r = MAIR_EL3;
        otherwise unreachable();
    return r;

// MAIR[] - non-assignment form
// -----------------------------------------------

MAIRType MAIR[]
    return MAIR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/sysregisters/MAIRType

```plaintext
type MAIRType;
```
// SCTLR[] - non-assignment form
// =============================
SCTLRType SCTLR[bits(2) regime]
bits(64) r;
case regime of
  when EL1  r = SCTLR_EL1;
  when EL2  r = SCTLR_EL2;
  when EL3  r = SCTLR_EL3;
  otherwise unreachable();
return r;

// SCTLR[] - non-assignment form
// =============================
SCTLRType SCTLR[]
return SCTLR[S1TranslationRegime()];

Library pseudocode for aarch64/functions/sysregisters/VBAR

// VBAR[] - non-assignment form
// =============================
bits(64) VBAR[bits(2) regime]
bits(64) r;
case regime of
  when EL1  r = VBAR_EL1;
  when EL2  r = VBAR_EL2;
  when EL3  r = VBAR_EL3;
  otherwise unreachable();
return r;

// VBAR[] - non-assignment form
// =============================
bits(64) VBAR[]
return VBAR[S1TranslationRegime()];

Library pseudocode for aarch64/functions/system/AArch64.CheckAdvSIMDFPSystemRegisterTraps

// Checks if an AArch64 MSR, MRS or SYS instruction on a SIMD or floating-point
// register is trapped under the current configuration. Returns a boolean which
// is TRUE if trapping occurs, plus a binary value that specifies the Exception
// level trapped to.
(boole, bits(2)) AArch64.CheckAdvSIMDFPSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bit read);

Library pseudocode for aarch64/functions/system/AArch64.CheckSVESystemRegisterTraps

// Checks if an AArch64 MSR/MRS/SYS instruction on a Scalable Vector
// register is trapped under the current configuration
(boole, bits(2)) AArch64.CheckSVESystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bit read);
AArch64.CheckSystemAccess()  
// Checks if an AArch64 MSR, MRS or SYS instruction is UNALLOCATED or trapped at the current exception level, security state and configuration, based on the opcode's encoding.  
boolean unallocated = FALSE;  
boolean need_secure = FALSE;  
bits(2) min_EL;  

// Check for traps by HCR_EL2.TIDCP  
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && HCR_EL2.TIDCP == 1 && op0 == 'x1' && crn == '1x1' then  
  // At EL0, it is IMPLEMENTATION_DEFINED whether attempts to execute system register access instructions with reserved encodings are trapped to EL2 or UNDEFINED  
  rcs_el0_trap = boolean IMPLEMENTATION_DEFINED "Reserved Control Space EL0 Trapped";  
  if PSTATE.EL == EL1 || rcs_el0_trap then  
    AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);  

// Check for unallocated encodings  
case op1 of  
  when '00x', '010'  
    min_EL = EL1;  
  when '011'  
    min_EL = EL0;  
  when '100'  
    min_EL = EL2;  
  when '101'  
    if !HaveVirtHostExt() then UNDEFINED;  
    min_EL = EL2;  
  when '110'  
    min_EL = EL3;  
  when '111'  
    min_EL = EL1;  
  need_secure = TRUE;  
  if UInt(PSTATE.EL) < UInt(min_EL) then  
    // Check for traps on read/write access to registers named _EL2, _EL02, _EL12 from non-secure EL1 when HCR_EL2.NV bit is set  
    nv_access = HaveNVExt() && min_EL == EL2 && PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.NV == '1';  
    if !nv_access then  
      UNDEFINED;  
    elsif need_secure && !IsSecure() then  
      UNDEFINED;  
    elsif AArch64.CheckUnallocatedSystemAccess(PSTATE.EL, op0, op1, crn, crm, op2, read) then  
      UNDEFINED;  

// Check for traps on accesses to SIMD or floating-point registers  
(take_trap, target_el) = AArch64.CheckAdvSIMDFPSystemRegisterTraps(op0, op1, crn, crm, op2, read);  
if take_trap then  
  AArch64.AdvSIMDFPAccessTrap(target_el);  

// Check for traps on accesses to Scalable Vector registers  
(take_trap, target_el) = AArch64.CheckSVESystemRegisterTraps(op0, op1, crn, crm, op2);  
if take_trap then  
  SVEAccessTrap(target_el);  

// Check for traps on access to all other system registers  
(take_trap, target_el) = AArch64.CheckSystemRegisterTraps(op0, op1, crn, crm, op2, read);  
if take_trap then  
  AArch64.SystemRegisterTrap(target_el, op0, op2, op1, crn, rt, crm, read);  

AArch64.CheckUnallocatedSystemAccess(EL1, op0, op2, op1, crn, crm, op2, read);  

// Checks if an AArch64 MSR, MRS or SYS instruction on a system register is trapped under the current configuration. Returns a boolean which is TRUE if trapping occurs, plus a binary value that specifies the Exception level trapped to.  
(boolean, bits(2)) AArch64.CheckSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
// Checks if an AArch64 MSR, MRS or SYS instruction is unallocated under the current configuration.
boolean AArch64.CheckUnallocatedSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bit read);

// AArch64.ExecutingATS1xPInstr()
// -----------------------------------------------
// Return TRUE if current instruction is AT S1E1R/WP
boolean AArch64.ExecutingATS1xPInstr()
{
  if !HavePrivATExt() then return FALSE;
  instr = ThisInstr();
  if instr<22+:10> == '1101010100' then
    op1 = instr<16+:3>;
    CRn = instr<12+:4>;
    CRm = instr<8+:4>;
    op2 = instr<5+:3>;
    return op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000','001'};
  else
    return FALSE;
}

// AArch64.ExecutingBROrBLROrRetInstr()
// -------------------------------------
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].
boolean AArch64.ExecutingBROrBLROrRetInstr()
{
  if !HaveBTIExt() then return FALSE;
  instr = ThisInstr();
  if instr<31:25> == '1101011' && instr<20:16> == '11111' then
    opc = instr<24:21>;
    return opc != '0101';
  else
    return FALSE;
}

// AArch64.ExecutingBTIInstr()
// ---------------------------
// Returns TRUE if current instruction is a BTI.
boolean AArch64.ExecutingBTIInstr()
{
  if !HaveBTIExt() then return FALSE;
  instr = ThisInstr();
  if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
    CRm = instr<11:8>;
    op2 = instr<7:5>;
    return (CRm == '0100' && op2<0> == '0');
  else
    return FALSE;
}

// AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);

// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
library pseudocode for aarch64/functions/system/AArch64.SysInstrWithResult

// Execute a system instruction with read (result operand).
// Returns the result of the instruction.

bits(64) AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2);

library pseudocode for aarch64/functions/system/AArch64.SysRegRead

// Read from a system register and return the contents of the register.

bits(64) AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2);

library pseudocode for aarch64/functions/system/AArch64.SysRegWrite

// Write to a system register.

AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);

library pseudocode for aarch64/functions/system/BTypeCompatible

boolean BTypeCompatible;

library pseudocode for aarch64/functions/system/BTypeCompatible_BTI

// BTypeCompatible_BTI
// ================
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)

  case hintcode of
    when '00'
      return FALSE;
    when '01'
      return PSTATE.BTYPE != '11';
    when '10'
      return PSTATE.BTYPE != '10';
    when '11'
      return TRUE;

library pseudocode for aarch64/functions/system/BTypeCompatible_PACIXSP

// BTypeCompatible_PACIXSP()
// ================
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()

  if PSTATE.BTYPE IN {'01', '10'} then
    return TRUE;
  elsif PSTATE.BTYPE == '11' then
    index = if PSTATE.EL == EL0 then 35 else 36;
    return SCTLR[index] == '0';
  else
    return FALSE;

library pseudocode for aarch64/functions/system/BTypeNext

bits(2) BTypeNext;

library pseudocode for aarch64/functions/system/InGuardedPage

boolean InGuardedPage;
// AAch64.ExceptionReturn()
// --------------------------------
AAch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTRL[] . IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
        // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' && spsr<4> == '1' && spsr<20> == '0' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then                // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32
        else                                     // Return to AArch64
            // ELR_ELx[63:56] might include a tag
            new_pc = AAch64.BranchAddr(new_pc);
    if UsingAArch32() then
        // 32 most significant bits are ignored.
        BranchTo(new_pc<31:0>, BranchType_ERET);
    else
        BranchToAddr(new_pc, BranchType_ERET);

Library pseudocode for aarch64/instrs/countop/CountOp

    enumeration CountOp     {
        CountOp_CLZ,
        CountOp_CLS,
        CountOp_CNT};

Library pseudocode for aarch64/instrs/extendreg/DecodeRegExtend

    // DecodeRegExtend()
    // -------------------
    // Decode a register extension option
    ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTX;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
Library pseudocode for aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType type, integer shift)
assert shift >= 0 && shift <= 4;
bits(N) val = X[reg];
boolean unsigned;
integer len;
case type of
    when ExtendType_SXTB unsigned = FALSE; len = 8;
    when ExtendType_SXTH unsigned = FALSE; len = 16;
    when ExtendType_SXTW unsigned = FALSE; len = 32;
    when ExtendType_SXTX unsigned = FALSE; len = 64;
    when ExtendType_UXTB unsigned = TRUE;  len = 8;
    when ExtendType_UXTH unsigned = TRUE;  len = 16;
    when ExtendType_UXTW unsigned = TRUE;  len = 32;
    when ExtendType_UXTX unsigned = TRUE;  len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

Library pseudocode for aarch64/instrs/extendreg/ExtendType

```
enumeration ExtendType  {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

Library pseudocode for aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
enumeration FPMaxMinOp  {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                         FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

Library pseudocode for aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

```
enumeration FPUnaryOp   {FPUnaryOp_ABS, FPUnaryOp_MOV,
                         FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

Library pseudocode for aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                         FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF,
                         FPConvOp_CVT_FtoI_US}
```
// BFXPreferred()
// ================
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
// DecodeBitMasks()
// ================

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)

bits(64) tmask, wmask;
bits(6) tmask_and, wmask_and;
bits(6) tmask_or, wmask_or;
bits(6) levels;

// Compute log2 of element size
// 2^len must be in range [2, M]
len = HighestSetBit(immN:NOT(imms));
if len < 1 then UNDEFINED;
assert M >= (1 << len);

// Determine S, R and S - R parameters
levels = ZeroExtend(Ont(l), 6);

// For logical immediates an all-ones value of S is reserved
// since it would generate a useless all-ones result (many times)
if immediate && (imms AND levels) == levels then
  UNDEFINED;
S = Uint(imms AND levels);
R = Uint(immr AND levels);
diff = S - R;    // 6-bit subtract with borrow

// From a software perspective, the remaining code is equivalent to:
// esize = 1 << len;
// d = Uint(diff<len-1:0>);
// welem = ZeroExtend(Ones(S + 1), esize);
// telem = ZeroExtend(Ones(d + 1), esize);
// wmask = Replicate(ROR(welem, R));
// tmask = Replicate(telem);
// return (wmask, tmask);

// Compute "top mask"
tmask_and = diff<5:0> OR NOT(levels);
tmask_or  = diff<5:0> AND levels;

// optimization of first step:
// tmask = Replicate(tmask_and<0> : '1', 32);

// Compute "wraparound mask"
wmask_and = immr OR NOT(levels);
wmask_or  = immr AND levels;

wmask = Zeros(64);
wmask = ((wmask

}
AND  Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
OR  Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));

// optimization of first step:
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
AND  Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
OR  Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));

wmask = ((wmask
AND  Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
OR  Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));

wmask = ((wmask
AND  Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
OR  Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));

wmask = ((wmask
AND  Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
OR  Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));

wmask = ((wmask
AND  Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
OR  Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

if diff<6> != '0' then // borrow from S - R
wmask = wmask AND tmask;
else
  wmask = wmask OR tmask;
return (wmask<M-1:0>, tmask<M-1:0>);

Library pseudocode for aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp


Library pseudocode for aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

// MoveWidePreferred()
// ---------------------
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<->ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
integer S = UInt(imms);
integer R = UInt(immr);
integer width = if sf == '1' then 64 else 32;

// element size must equal total immediate size
if sf == '1' && immN:imms != '1xxxxxx' then return FALSE;
if sf == '0' && immN:imms != '00xxxxx' then return FALSE;

// for MOVZ must contain no more than 16 ones
if S < 16 then
  // ones must not span halfword boundary when rotated
  return (~R MOD 16) <= (15 - S);
else
  // for MOVZ must contain no more than 16 zeros
  if S >= width - 15 then
    // zeros must not span halfword boundary when rotated
    return (R MOD 16) <= (S - (width - 15));
  return FALSE;
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00' return ShiftType_LSL;
        when '01' return ShiftType_LSR;
        when '10' return ShiftType_ASR;
        when '11' return ShiftType_ROR;

// ShiftReg()
// =========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType type, integer amount)
    bits(N) result = X[reg];
    case type of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;

// ShiftType

enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};

// LogicalOp

enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};

// MemAtomicOp


// MemOp

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
// Prefetch()
// =========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)

    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ; // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC; // PLI: preload instructions
        when '10' hint = Prefetch_WRITE; // PST: prepare for store
        when '11' return;                 // unallocated hint

    target = UInt(prfop<2:1>);       // target cache level
    stream = (prfop<0> != '0');      // streaming (non-temporal)

    Hint_Prefetch(address, hint, target, stream);
    return;

Library pseudocode for aarch64/instrs/system/barriers/barrierop/MemBarrierOp

enumeration MemBarrierOp { MemBarrierOp_DSB         // Data Synchronization Barrier
                        , MemBarrierOp_DMB         // Data Memory Barrier
                        , MemBarrierOp_ISB         // Instruction Synchronization Barrier
                        , MemBarrierOp_SSSBB       // Speculative Synchronization Barrier to VA
                        , MemBarrierOp_PSSBB       // Speculative Synchronization Barrier to PA
                        , MemBarrierOp_SB          // Speculation Barrier
                        };
// SysOp()  
// ========

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2) {
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;  // S1E1R
        when '100 0111 1000 000' return Sys_AT;  // S1E2R
        when '110 0111 1000 000' return Sys_AT;  // S1E3R
        when '000 0111 1000 001' return Sys_AT;  // S1E1W
        when '100 0111 1000 001' return Sys_AT;  // S1E2W
        when '110 0111 1000 001' return Sys_AT;  // S1E3W
        when '000 0111 1000 010' return Sys_AT;  // S1E0R
        when '000 0111 1000 011' return Sys_AT;  // S1E0W
        when '100 0111 1000 100' return Sys_AT;  // S12E1R
        when '100 0111 1000 101' return Sys_AT;  // S12E1W
        when '100 0111 1000 110' return Sys_AT;  // S12E0R
        when '100 0111 1000 111' return Sys_AT;  // S12E0W
        when '011 0111 0100 001' return Sys_DC;  // ZVA
        when '000 0111 0110 001' return Sys_DC;  // IVAC
        when '000 0111 0110 010' return Sys_DC;  // ISW
        when '011 0111 1010 001' return Sys_DC;  // CVAC
        when '000 0111 1010 010' return Sys_DC;  // CSW
        when '011 0111 1010 101' return Sys_DC;  // CVAU
        when '000 0111 1010 110' return Sys_DC;  // CISW
        when '000 0111 0000 000' return Sys_IC;  // IALLUIS
        when '000 0111 0100 000' return Sys_IC;  // IALLU
        when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
        when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
        when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
        when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
        when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
        when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
        when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
        when '000 1000 0011 100' return Sys_TLBI; // ASIDE1IS
        when '000 1000 0011 101' return Sys_TLBI; // VAAE1IS
        when '100 1000 0011 101' return Sys_TLBI; // VAE2IS
        when '110 1000 0011 101' return Sys_TLBI; // VALLE1IS
        when '000 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
        when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
        when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
        when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
        when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
        when '100 1000 0111 000' return Sys_TLBI; // ALLE2
        when '110 1000 0111 000' return Sys_TLBI; // ALLE3
        when '000 1000 0111 001' return Sys_TLBI; // VAE1
        when '100 1000 0111 001' return Sys_TLBI; // VAE2
        when '110 1000 0111 001' return Sys_TLBI; // VAE3
        when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI; // VAAE1
        when '100 1000 0111 011' return Sys_TLBI; // VAE2
        when '110 1000 0111 011' return Sys_TLBI; // VALLE1
        when '000 1000 0111 100' return Sys_TLBI; // VMALLE1
        when '100 1000 0111 100' return Sys_TLBI; // ALLE2
        when '110 1000 0111 100' return Sys_TLBI; // ALLE3
        when '000 1000 0111 101' return Sys_TLBI; // VAE1
        when '100 1000 0111 101' return Sys_TLBI; // VAE2
        when '110 1000 0111 101' return Sys_TLBI; // VALLE1
        when '000 1000 0111 110' return Sys_TLBI; // VMALLS12E1
        when '000 1000 0111 111' return Sys_TLBI; // VAALE1
return Sys_SYS;
}
enum VBitOp {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};


// Reduce()
// ---------

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;
    if N == esize then
        return input<esize-1:0>;
    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize);
    lo = Reduce(op, input<half-1:0>, esize);
    case op of
        when ReduceOp_FMINNUM result = FPMinNum(lo, hi, FPCR);
        when ReduceOp_FMAXNUM result = FPMaxNum(lo, hi, FPCR);
        when ReduceOp_FMIN result = FPMin(lo, hi, FPCR);
        when ReduceOp_FMAX result = FPMax(lo, hi, FPCR);
        when ReduceOp_FADD result = FPAdd(lo, hi, FPCR);
        when ReduceOp_ADD result = lo + hi;
    return result;

// AArch64.InstructionDevice()
// ===========================================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                           bits(52) ipaddress, integer level,
                                           AccType acctype, boolean iswrite, boolean secondstage,
                                           boolean s2fs1walk)

  c = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
  assert c IN {Constraint_NONE, Constraint_FAULT};

  if c == Constraint_FAULT then
    addrdesc.fault = AArch64.PermissionFault(ipaddress, bit UNKNOWN, level, acctype, iswrite,
                                             secondstage, s2fs1walk);
  else
    addrdesc.memattrs.type = MemType_Normal;
    addrdesc.memattrs.inner.attrs = MemAttr_NC;
    addrdesc.memattrs.inner.hints = MemHint_No;
    addrdesc.memattrs.outer = addrdesc.memattrs.inner;
    addrdesc.memattrs.tagged = FALSE;
    addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

  return addrdesc;
library pseudocode for aarch64/translation/attrs/AArch64.S1AttrDecode

// AArch64.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;
    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    memattrs.tagged = FALSE;
    if ((attrfield<7:4> != '0000' && attrfield<7:4> != '1111' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);
    if !HaveMTEExt() && attrfield<7:4> == '1111' && attrfield<3:0> == '0000' then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);
    if attrfield<7:4> == '0000' then            // Device
        memattrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000'  memattrs.device = DeviceType_nGnRnE;
            when '0100'  memattrs.device = DeviceType_nGnRE;
            when '1000'  memattrs.device = DeviceType_GRE;
            when '1100'  memattrs.device = DeviceType_GRE;
            otherwise  Unreachable();         // Reserved, handled above
    elsif attrfield<3:0> != '0000'  then        // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    elsif HaveMTEExt() && attrfield == '11110000' then // Tagged, Normal
        memattrs.tagged = TRUE;
        memattrs.type = MemType_Normal;
        memattrs.outer.attrs = MemAttr WB;
        memattrs.inner.attrs = MemAttr WB;
        memattrs.outer.hints = MemHint_RWA;
        memattrs.inner.hints = MemHint_RWA;
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    else
        Unreachable();                          // Reserved, handled above
    return MemAttrDefaults(memattrs);
// AArch64.TranslateAddressS1Off()
// ===============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
assert !EUUsingAArch32(S1TranslationRegime());

TLBRecord result;
Top = AddrTop(vaddress, (acctype == AccType_IFETCH), PSTATE.EL);
if !IsZero(vaddress<Top:PAMax>()) then
  level = 0;
ipaddress = bits(52) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;
result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,bit UNKNOWN, level, acctype, iswrite, secondstage, s2fs1walk);
return result;
default_cacheable = (HasS2Translation() && HCR_EL2.DC == '1');
if default_cacheable then
  // Use default cacheable settings
  result.addrdesc.memattrs.type = MemType_Normal;
  result.addrdesc.memattrs.inner.attrs = MemAttr_WB;  // Write-back
  result.addrdesc.memattrs.inner.hints = MemHint_FMA;
  result.addrdesc.memattrs.shareable = FALSE;
  result.addrdesc.memattrs.outershareable = FALSE;
  result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
elsif acctype != AccType_IFETCH then
  // Treat data as Device
  result.addrdesc.memattrs.type = MemType_Device;
  result.addrdesc.memattrs.device = DeviceType_nGnRnE;
  result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
  result.addrdesc.memattrs.tagged = FALSE;
else
  // Instruction cacheability controlled by SCTLR_ELx.I
  cacheable = SCTLR[]:I == '1';
  result.addrdesc.memattrs.type = MemType_Normal;
  if cacheable then
    result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
    result.addrdesc.memattrs.inner.hints = MemHint_RA;
  else
    result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
    result.addrdesc.memattrs.inner.hints = MemHint_No;
  result.addrdesc.memattrs.shareable = TRUE;
  result.addrdesc.memattrs.outershareable = TRUE;
  result.addrdesc.memattrs.tagged = FALSE;
result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;
result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

result.perm.ap = bits(3) UNKNOWN;
result.perm.xn = '0';
result.perm.pxn = '0';
result.nG = bit UNKNOWN;
result.contiguous = boolean UNKNOWN;
result.domain = bits(4) UNKNOWN;
result.level = integer UNKNOWN;
result.blocksize = integer UNKNOWN;
result.addrdesc.paddress.address = vaddress<51:0>;
result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
result.addrdesc.fault = AArch64.NoFault();
return result;
/** AArch64.AccessIsPrivileged()
     *  ------------------------------

     * boolean AArch64.AccessIsPrivileged(AccType acctype)
     * 
     * el = AArch64.AccessUsesEL(acctype);
     * if el == EL0 then
     *   ispriv = FALSE;
     * elsif el == EL3 then
     *   ispriv = TRUE;
     * elsif el == EL2 && (!IsInHost() || HCR_EL2.TGE == '0') then
     *   ispriv = TRUE;
     * elsif HaveUAOExt() && PSTATE.UAO == '1' then
     *   ispriv = TRUE;
     * else
     *   ispriv = (acctype != AccType_UNPRIV);
     * return ispriv;

/** AArch64.AccessUsesEL()
     * -----------------------

     * // Returns the Exception Level of the regime that will manage the translation for a given access type.
     * bits(2) AArch64.AccessUsesEL(AccType acctype)
     * 
     * if acctype == AccType_UNPRIV then
     *   return EL0;
     * elsif acctype == AccType_NV2REGISTER then
     *   return EL2;
     * else
     *   return PSTATE.EL;


// AArch64.CheckPermission()
// ---------------------------------
// Function used for permission checking from AArch64 stage 1 translations

FaultRecord AArch64.CheckPermission(Permissions perms, bits(64) vaddress, integer level, bit NS, AccType accctype, boolean iswrite)
assert !ELUsingAArch32(SITranslationRegime());

wxn = SCTLR[].WXN == '1';
if (PSTATE.EL == EL0 ||
    IsInHost() ||
    (PSTATE.EL == EL1 && !HaveNV2Ext()) ||
    (PSTATE.EL == EL1 && HaveNV2Ext() && (accctype == AccType_NV2REGISTER || !ELIsInHost(EL2))))

priv_r = TRUE;
priv_w = perms.ap<2> == '0';
user_r = perms.ap<1> == '1';
user_w = perms.ap<2:1> == '01';

ispriv = AArch64.AccessIsPrivileged(accctype);

if ispriv then
    (r, w, xn) = (priv_r, priv_w, priv_xn);
else
    (r, w, xn) = (user_r, user_w, user_xn);
else
    // Access from EL2 or EL3
    r = TRUE;
    w = perms.ap<2> == '0';
    xn = perms.xn == '1' || (w && wxn);

// Restriction on Secure instruction fetch
if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
    xn = TRUE;

if accctype == AccType_IFETCH then
    fail = xn;
    failedread = TRUE;
elsif accctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW } then
    fail = !r || !w;
    failed = !r;
elsif iswrite then
    fail = !w;
    failed = FALSE;
elsif accctype == AccType_DC && PSTATE.EL != EL0 then
    // DC maintenance instructions operating by VA, cannot fault from stage 1 translation,
    // other than DC IVAC, which requires write permission, and operations executed at EL0,
    // which require read permission.
    fail = FALSE;
else
    fail = !r;
    failedread = TRUE;
if fail then
    secondstage = FALSE;
s2fs1walk = FALSE;
ipaddress = bits(52) UNKNOWN;
return AArch64.PermissionFault(ipaddress, bit UNKNOWN, level, acctype, !failedread, secondstage, s2fs1walk);
else
return AArch64.NoFault();

Library pseudocode for aarch64/translation/checks/AArch64.CheckS2Permission

// AArch64.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(52) ipaddress, Integer level, AccType acctype, boolean iswrite, bit NS, boolean s2fs1walk, boolean hwupdatewalk)
assert IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2)) && HasS2Translation();
r = perms.ap<1> == '1';
w = perms.ap<2> == '1';
if HaveExtendedExecuteNeverExt() then
case perms.xn:perms.xxn of
  when '00'  xn = FALSE;
  when '01'  xn = PSTATE.EL == EL1;
  when '10'  xn = TRUE;
  when '11'  xn = PSTATE.EL == EL0;
else
  xn = perms.xn == '1';
// Stage 1 walk is checked as a read, regardless of the original type
if acctype == AccType_IFETCH && !s2fs1walk then
  fail = xn;
  failedread = TRUE;
elsif (acctype IN {AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW}) && !s2fs1walk then
  fail = !r || !w;
  failedread = !r;
elsif iswrite && !s2fs1walk then
  fail = !w;
  failedread = !iswrite;
elsif acctype == AccType_DC && PSTATE.EL != EL0 && !s2fs1walk then
  // DC maintenance instructions operating by VA, with the exception of DC IVAC, do
  // not generate Permission faults from stage 2 translation, other than when
  // performing a stage 1 translation table walk.
  fail = FALSE;
elsif hwupdatewalk then
  fail = !w;
  failedread = !iswrite;
else
  fail = !r;
  failedread = !iswrite;
if fail then
  domain = bits(4) UNKNOWN;
  secondstage = TRUE;
  return AArch64.PermissionFault(ipaddress, NS, level, acctype, !failedread, secondstage, s2fs1walk);
else
  return AArch64.NoFault();

Shared Pseudocode Functions
Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

// AArch64.CheckBreakpoint()
// ------------------------
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, AccType acctype, integer size)
assert !ELUsingAArch32(SITranslationRegime());
assert (UsingAArch32() && size IN {2,4}) || size == 4;
match = FALSE;
for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
    match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
    match = match || match_i;
if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Breakpoint;
    Halt(reason);
elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    return AArch64.DebugFault(acctype, iswrite);
else
    return AArch64.NoFault();

Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

// AArch64.CheckDebug()
// ---------------------
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)
FaultRecord fault = AArch64.NoFault();
d_side = (acctype != AccType_IFETCH);
generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
halt = HaltOnBreakpointOrWatchpoint();
if generate_exception || halt then
    if d_side then
        fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
    else
        fault = AArch64.CheckBreakpoint(vaddress, acctype, size);
else
    return fault;
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)
assert !ELUsingAArch32(S1TranslationRegime());
match = FALSE;
ispriv = AArch64.AccessIsPrivileged(acctype);
for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite);
if match && HaltOnBreakpointOrWatchpoint() then
if acctype != AccType_NONFAULT && acctype != AccType_CNOTFIRST then
reason = DebugHalt_Watchpoint;
Halt(reason);
else
// Fault will be reported and cancelled
return AArch64.DebugFault(acctype, iswrite);
elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
return AArch64.DebugFault(acctype, iswrite);
else
return AArch64.NoFault();
end

Library pseudocode for aarch64/translation/faults/AArch64.AccessFlagFault

// AArch64.AccessFlagFault()
// =========================

FaultRecord AArch64.AccessFlagFault(bits(52) ipaddress, bit NS, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)
extflag = bit UNKNOWN;
errortype = bits(2) UNKNOWN;
return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress, NS, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.AddressSizeFault

// AArch64.AddressSizeFault()
// =========================

FaultRecord AArch64.AddressSizeFault(bits(52) ipaddress, bit NS, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)
extflag = bit UNKNOWN;
errortype = bits(2) UNKNOWN;
return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress, NS, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
Library pseudocode for aarch64/translation/faults/AArch64.AlignmentFault

// AArch64.AlignmentFault()
// ================

FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

  ipaddress = bits(52) UNKNOWN;
  level = integer UNKNOWN;
  extflag = bit UNKNOWN;
  errortype = bits(2) UNKNOWN;
  s2fs1walk = boolean UNKNOWN;

  return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress, bit UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.AsynchExternalAbort

// AArch64.AsynchExternalAbort()
// ================

// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

  type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
  ipaddress = bits(52) UNKNOWN;
  level = integer UNKNOWN;
  acctype = AccType_NORMAL;
  iswrite = boolean UNKNOWN;
  secondstage = FALSE;
  s2fs1walk = FALSE;

  return AArch64.CreateFaultRecord(type, ipaddress, bit UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.DebugFault

// AArch64.DebugFault()
// ================

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

  ipaddress = bits(52) UNKNOWN;
  errortype = bits(2) UNKNOWN;
  level = integer UNKNOWN;
  extflag = bit UNKNOWN;
  secondstage = FALSE;
  s2fs1walk = FALSE;

  return AArch64.CreateFaultRecord(Fault_Debug, ipaddress, bit UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
Library pseudocode for aarch64/translation/faults/AArch64.NoFault

```plaintext
// AArch64.NoFault()
// ================
FaultRecord AArch64.NoFault()
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
    return AArch64.CreateFaultRecord(Fault_None, ipaddress, bit UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.PermissionFault

```plaintext
// AArch64.PermissionFault()
// =========================
FaultRecord AArch64.PermissionFault(bits(52) ipaddress, bit NS, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Permission, ipaddress, NS, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.TranslationFault

```plaintext
// AArch64.TranslationFault()
// ==========================
FaultRecord AArch64.TranslationFault(bits(52) ipaddress, bit NS, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Translation, ipaddress, NS, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
```
// AArch64.CheckAndUpdateDescriptor()
// ---------------------------------
// Check and update translation table descriptor if hardware update is configured

FaultRecord AArch64.CheckAndUpdateDescriptor(DescriptorUpdate result, FaultRecord fault,
    boolean secondstage, bits(64) vaddress, AccType acctype,
    boolean iswrite, boolean s2fs1walk, boolean hwupdatewalk)

    // Check if access flag can be updated
    // Address translation instructions are permitted to update AF but not required
    if result.AF then
        if fault.type == Fault_None then
            hw_update_AF = TRUE;
        elsif ConstrainUnpredictable(Unpredictable_AFUPDATE) == Constraint_TRUE then
            hw_update_AF = TRUE;
        else
            hw_update_AF = FALSE;
    if result.AP && fault.type == Fault_None then
        write_perm_req = (iswrite || acctype IN {AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW}) && !s2fs1walk;
        hw_update_AP = (write_perm_req && !(acctype IN {AccType_AT, AccType_DC, AccType_DC_UNPRIV})) || hwupdatewalk;
    else
        hw_update_AP = FALSE;
    if hw_update_AF || hw_update_AP then
        if secondstage || !HasS2Translation() then
            descaddr2 = result.descaddr;
        else
            hwupdatewalk = TRUE;
            descaddr2 = AArch64.SecondStageWalk(result.descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
            if IsFault(descaddr2) then
                return descaddr2.fault;
        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        desc = _Mem[descaddr2, 8, accdesc];
        el = AArch64.AccessUsesEL(acctype);
        case el of
            when EL3
                reversedescriptors = SCTLR_EL3.EE == '1';
            when EL2
                reversedescriptors = SCTLR_EL2.EE == '1';
            otherwise
                reversedescriptors = SCTLR_EL1.EE == '1';
            if reversedescriptors then
                desc = BigEndianReverse(desc);
            if hw_update_AF then
                desc<10> = '1';
            if hw_update_AP then
                desc<7> = (if secondstage then '1' else '0');
            _Mem[descaddr2,8,accedesc] = if reversedescriptors then BigEndianReverse(desc) else desc;
        return fault;
}
// AArch64.FirstStageTranslate()
// -----------------------------------------------
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
  s1_enabled = SCTLR_EL2.M == '1';
elsif HasS2Translation() then
  s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
else
  s1_enabled = SCTLR[].M == '1';

ipaddress = bits(52) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;

if s1_enabled then                          // First stage enabled
  S1 = AArch64.TranslationTableWalk(ipaddress, '1', vaddress, acctype, iswrite, secondstage, s2fs1walk, size);
  permissioncheck = TRUE;
  if acctype == AccType_IFETCH then
    InGuardedPage = S1.GP == '1';      // Global state updated on instruction fetch that denotes
    // if the fetched instruction is from a guarded page.
  else
    S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
    permissioncheck = FALSE;
  if UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() then
    if S1.addrdesc.memattrs.type == MemType_Device && S1.addrdesc.memattrs.device != DeviceType_GRE then
      nTLSMD = if S1TranslationRegime() == EL2 then SCTLR_EL2.nTLSMD else SCTLR_EL1.nTLSMD;
      if nTLSMD == '0' then
        S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
      // Check for unaligned data accesses to Device memory
      if (!wasaligned && acctype != AccType_IFETCH) || (acctype != AccType_DCZVA))
        & S1.addrdesc.memattrs.type == MemType_Device && !IsFault(S1.addrdesc) then
          S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
        if !IsFault(S1.addrdesc) && permissioncheck then
          S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level, S1.addrdesc.paddress.NS, acctype, iswrite);
      // Check for instruction fetches from Device memory not marked as execute-never. If there has
      // not been a Permission Fault then the memory is not marked execute-never.
      if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
          acctype == AccType_IFETCH) then
        S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level, acctype, iswrite, secondstage, s2fs1walk);
      // Check and update translation table descriptor if required
      hwupdatewalk = FALSE;
      s2fs1walk = FALSE;
      S1.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S1.descupdate, S1.addrdesc.fault, secondstage, vaddress, acctype, iswrite, s2fs1walk, hwupdatewalk);
    return S1.addrdesc;
/ AArch64.FullTranslate()
// -----------------------------------
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.
AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
  boolean wasaligned, integer size)

// First Stage Translation
S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
if !IsFault(S1) && !HaveNV2Ext() && acctype == AccType_NV2REGISTER) && HasS2Translation()
  s2fs1walk = FALSE;
  hwupdatewalk = FALSE;
  result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
    size, hwupdatewalk);
else
  result = S1;
return result;
AAAch64.SecondStageTranslate()  
// AArch64.SecondStageTranslate()  
// ==============================  
// Perform a stage 2 translation walk. The function used by Address Translation operations is  
// similar except it uses the translation regime specified for the instruction.  

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress,  
    AccType acctype, boolean iswrite, boolean wasaligned,  
    boolean s2fs1walk, integer size, boolean hwupdatewalk)

assert HasS2Translation();

S2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';  
secondstage = TRUE;

if S2_enabled then  // Second stage enabled  
    ipaddress = S1.paddress.address<51:0>;  
    NS = S1.paddress.NS;
    S2 = AArch64.TranslationTableWalk(ipaddress, NS, vaddress, acctype, iswrite, secondstage,  
        s2fs1walk, size);

    // Check for unaligned data accesses to Device memory
    if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA)  
        && S2.addrdesc.memattrs.type == MemType_Device && !IsFault(S2.addrdesc) then
        S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

    // Check for permissions on Stage2 translations
    if !IsFault(S2.addrdesc) then
        S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,  
            acctype, iswrite, NS,s2fs1walk, hwupdatewalk);

    // Check for instruction fetches from Device memory not marked as execute-never. As there  
    // has not been a Permission Fault then the memory is not marked execute-never.
    if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device  
        && acctype == AccType_IFETCH) then
        S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,  
            acctype, iswrite, secondstage, s2fs1walk);

    // Check for protected table walk
    if (s2fs1walk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1'  
        && S2.addrdesc.memattrs.type == MemType_Device) then
        S2.addrdesc.fault = AArch64.PermissionFault(ipaddress, S1.paddress.NS, S2.level, acctype,  
            iswrite, secondstage, s2fs1walk);

    // Check and update translation table descriptor if required
    S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault,  
        secondstage, vaddress, acctype,  
        iswrite, s2fs1walk, hwupdatewalk);

        result = CombineS1S2Desc(S1, S2.addrdesc);
    else  
        result = S1;

    return result;
Library pseudocode for aarch64/translation/translation/AArch64.SecondStageWalk

```c
// AArch64.SecondStageWalk()
// -----------------------------------
// Perform a stage 2 translation on a stage 1 translation page table walk access.
AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype, boolean iswrite, integer size, boolean hwupdatewalk)

assert HasS2Translation();

s2fs1walk = TRUE;
wasaligned = TRUE;
return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk, size, hwupdatewalk);
```

Library pseudocode for aarch64/translation/translation/AArch64.TranslateAddress

```c
// AArch64.TranslateAddress()
// --------------------------
// Main entry point for translating an address
AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
    result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(vaddress);

return result;
```
Library pseudocode for aarch64/translation/walk/AArch64.TranslationTableWalk
AArch64.TranslationTableWalk()
// ==============================
// Returns a result of a translation table walk
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(52) ipaddress, bit s1_nonsecure, bits(64) vaddress,
AccType acctype, boolean iswrite, boolean secondstage,
boolean s2fs1walk, integer size)

if !secondstage then
    assert !ElUsingAArch32(S1TranslationRegime());
else
    assert IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure() && !ElUsingAArch32(EL2) && HasS2Translation());

TLBRecord result;
AddressDescriptor descaddr;
bits(64) baseregister;
bits(64) inputaddr; // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2

descaddr.memattrs.type = MemType_Normal;

// Derived parameters for the page table walk:
//  grainsize = Log2(Size of Table) - Size of Table is 4KB, 16KB or 64KB in AArch64
//  stride = Log2(Address per Level) - Bits of address consumed at each level
//  firstblocklevel = First level where a block entry is allowed
//  ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx.VTCR_EL2.PS
//  inputsize = Log2(Size of Input Address) - Input Address size in bits
//  level = Level to start walk from
// This means that the number of levels after start level = 3-level

if !secondstage then
    // First stage translation
    inputaddr = ZeroExtend(vaddress);
    el = AArch64.AccessUsesEL(acctype);
    top = AddrTop(inputaddr, (acctype == AccType_IFETCH), el);
    if el == EL3 then
        largegrain = TCR_EL3.TG0 == '01';
        midgrain = TCR_EL3.TG0 == '10';
        inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
        inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
        if inputsize < inputsize_min then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_min;
        ps = TCR_EL3.PS;
        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top:inputsize>);
        disabled = FALSE;
        baseregister = TTBR0_EL3;
        descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0, secondstage);
        reversedescriptors = SCTLR_EL3.EE == '1';
        lookupsecure = TRUE;
        singlepriv = TRUE;
        update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
        hierattribdisabled = AArch64.HaveHPDExt() && TCR_EL3.HPD == '1';
    elsif ELIsInHost(el) then
        if inputaddr<top> == '0' then
            largegrain = TCR_EL2.TG0 == '01';
            midgrain = TCR_EL2.TG0 == '10';
            inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
            inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top>);
        disabled = FALSE;
disabled = TCR_EL2.EPD0 == '1' | (PSTATE.EL == EL0 && HaveE0PDExt() && TCR_EL2.E0PD0 = baseregister = TTBR0_EL2;
descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondstage);
hierattrdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD0 == '1';
else
inputsize = 64 - UInt(TCR_EL2.TLSZ);
largegrain = TCR_EL2.TG1 == '11'; // TG1 and TG0 encodings differ
midgrain = TCR_EL2.TG1 == '01';
inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
inputsize_min = 64 - (if HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
if inputsize < inputsize_min then
  c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
  assert c IN {Constraint_FORCE, Constraint_FAULT};
  if c == Constraint_FORCE then inputsize = inputsize_min;
elsefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputaddr<top:inputsize>);
baseline = TCR_EL2.E0PD0 == '1';
else
inputsize = 64 - UInt(TCR_EL2.TLSZ);
largegrain = TCR_EL2.TG1 == '11'; // TG1 and TG0 encodings differ
midgrain = TCR_EL2.TG1 == '01';
inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
inputsize_min = 64 - (if HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
if inputsize < inputsize_min then
  c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
  assert c IN {Constraint_FORCE, Constraint_FAULT};
  if c == Constraint_FORCE then inputsize = inputsize_min;
elsefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top:inputsize>);
baseline = TCR_EL2.E0PD0 == '1';
else
inputsize = 64 - UInt(TCR_EL2.TLSZ);
largegrain = TCR_EL2.TG1 == '11'; // TG1 and TG0 encodings differ
midgrain = TCR_EL2.TG1 == '01';
inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
inputsize_min = 64 - (if HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
if inputsize < inputsize_min then
  c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
  assert c IN {Constraint_FORCE, Constraint_FAULT};
  if c == Constraint_FORCE then inputsize = inputsize_min;
elsefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputaddr<top:inputsize>);
baseline = TCR_EL2.E0PD0 == '1';
else
inputsize = 64 - UInt(TCR_EL2.TLSZ);
largegrain = TCR_EL2.TG1 == '11'; // TG1 and TG0 encodings differ
midgrain = TCR_EL2.TG1 == '01';
inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
inputsize_min = 64 - (if HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
if inputsize < inputsize_min then
  c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
  assert c IN {Constraint_FORCE, Constraint_FAULT};
  if c == Constraint_FORCE then inputsize = inputsize_min;
c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
assert c IN (Constraint_FORCE, Constraint_FAULT);
if c == Constraint_FORCE then inputsize = inputsize_min;
basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputaddr);
disabled = TCR_EL1.EPD1 == '1' || (PSTATE.EL == EL0 && HaveE0PDExt() && TCR_EL1.E0PD1 == '1')
&& disabled || (el == EL0 && acctype == AccType_NONFAULT && TCR_EL1.NFD1 == '1')
baseregister = TTBR1_EL1;
descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1, secondstage)
&& hierattrdisabled = AArch64.HaveHPDExt() && TCR_EL1.HPD1 == '1';
ps = TCR_EL1.IPS;
reversedescriptors = SCTLR_EL1.EE == '1';
lookupsecure = IsSecure();
singlepriv = FALSE;
update_AF = HaveAccessFlagUpdateExt() && TCR_EL1.HA == '1';
update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL1.HD == '1';
if largegrain then
  grainsize = 16; // Log2(64KB page size)
  firstblocklevel = (if Have52BitPAExt() then 1 else 2); // Largest block is 4TB (2^42 bytes)
  stride = grainsize - 3; // Log2(page size / 8 bytes)
else if midgrain then
  grainsize = 14; // Log2(16KB page size)
  firstblocklevel = 2;
else // Small grain
  grainsize = 12; // Log2(4KB page size)
  firstblocklevel = 1;
  stride = grainsize - 3; // Log2(page size / 8 bytes)
// The starting level is the number of strides needed to consume the input address
level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));
else // Second stage translation
  inputaddr = ZeroExtend(ipaddress);
  // Stage 2 translation table walk for the Secure EL2 translation regime
  if IsSecureEL2Enabled() && IsSecure() then
    // Stage 2 translation walk is in the Non-secure IPA space or the Secure IPA space
    t0size = if sl_nonsecure == '1' then VTCR_EL2.TOSZ else VSTCR_EL2.TOSZ;
    tg0 = if sl_nonsecure == '1' then VTCR_EL2.TGO else VSTCR_EL2.TGO;
// Stage 2 translation table walk is to the Non-secure PA space or to the Secure PA space
    nswalk = if sl_nonsecure == '1' then VTCR_EL2.NSW else VSTCR_EL2.SW;
// Stage 2 translation accesses the Non-secure PA space or the Secure PA space
    if nswalk == '1' then
      nsaccess = '1'; // When walk is non-secure, access
    else
      if sl_nonsecure == '0' then
        nsaccess = VSTCR_EL2.SA; // When walk is secure and in the Non-secure IPA space
      else
        if VSTCR_EL2.SW == '1' || VSTCR_EL2.SA == '1' then nsaccess = '1'; // When walk is secure and in the Secure IPA space
        else nsaccess = VTCR_EL2.NSA; // When walk is secure and in the Non-secure IPA space
    else
      t0size = VTCR_EL2.TOSZ;
      tg0 = VTCR_EL2.TGO;
      nsaccess = '1';
inputsize  = 64 - UInt(t0size);
largegrain = tg0 == '01';
midgrain = tg0 == '10';
inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
if inputsize < inputsize_min then
  c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
  assert c IN (Constraint_FORCE, Constraint_FAULT);
  if c == Constraint_FORCE then inputsize = inputsize_min;
ps = VTCR_EL2.PS;
basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<63:inputsize);
disabled = FALSE;
descaddr.memattrs = WalkAttrDecode(VTCR_EL2.IRGNO, VTCR_EL2.ORGNO, VTCR_EL2.SHO, secondstage);
reversedescriptors = SCTLR_EL2.EE == '1';
singlepriv = TRUE;
update_AF = HaveAccessFlagUpdateExt() && VTCR_EL2.HA == '1';
update_AP = HaveDirtyBitModifierExt() && update_AF && VTCR_EL2.HD == '1';
lookupsecure = if IsSecureEL2Enabled() then s1_nonsecure == '0' else FALSE;
// Stage2 translation table walk is to secure PA space or to Non-secure PA space
baseregister = if lookupsecure then VSTTBR_EL2 else VTTBR_EL2;
startlevel = if lookupsecure then UInt(VSTCR_EL2.SL0) else UInt(VTCR_EL2.SL0);
if largegrain then
  grainsize = 16; // Log2(64KB page size)
  level = 3 - startlevel;
  firstblocklevel = (if Have52BitPAExt() then 1 else 2); // Largest block is 4TB (2^42 bytes)
    // and 512MB (2^29 bytes) otherwise
elsif midgrain then
  grainsize = 14; // Log2(16KB page size)
  level = 3 - startlevel;
  firstblocklevel = 2; // Largest block is 32MB (2^25 bytes)
else // Small grain
  grainsize = 12; // Log2(4KB page size)
  if HaveSmallPageTblExt() && startlevel == 3 then
    level = startlevel; // Startlevel 3 (VTCR_EL2.SL0 or VSCTR_EL2.SL0)
    // Largest block is 1GB (2^30 bytes)
  else
    level = 2 - startlevel;
    firstblocklevel = 1; // Largest block is 512MB (2^29 bytes)
  stride = grainsize - 3; // Log2(page size / 8 bytes)

// Limits on IPA controls based on implemented PA size. Level 0 is only
// supported by small grain translations
if largegrain then // 64KB pages
  // Level 1 only supported if implemented PA size is greater than 2^42 bytes
  if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
elsif midgrain then // 16KB pages
  // Level 1 only supported if implemented PA size is greater than 2^40 bytes
  if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
else // Small grain, 4KB pages
  // Level 0 only supported if implemented PA size is greater than 2^42 bytes
  if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;
// If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
inputsizecheck = inputsize;
if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
  case ConstraintUnpredictable(Unpredictable_LARGEIPA) of
    when Constraint_FORCE // Restrict the inputsize to the PAMax value
      inputsize = PAMax();
      inputsizecheck = PAMax();
    when Constraint_FORCENOSLCHECK // As FORCE, except use the configured inputsize in the size checks below
      inputsize = PAMax();
    when Constraint_FAULT // Generate a translation fault
      basefound = FALSE;
    otherwise
      Unreachable();
  endcase
// Number of entries in the starting level table =
// (Size of Input Address)/(Address per level)^(Num levels remaining)*(Size of Table)
startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)
// Check for starting level table with fewer than 2 entries or longer than 16 pages.
// Lower bound check is: startsizecheck < Log2(2 entries)
// Upper bound check is: startsizecheck > Log2(pagesize/8*16)
if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;
if !basefound || disabled then
  level = 0; // AArch32 reports this as a level 1 fault
  result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
return result;
case ps of
  when '000'  outputsize = 32;
  when '001'  outputsize = 36;
  when '010'  outputsize = 40;
  when '011'  outputsize = 42;
  when '100'  outputsize = 44;
  when '101'  outputsize = 48;
  when '110'  outputsize = (if Have52BitPAExt() && largegrain then 52 else 48);
  otherwise  outputsize = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

if outputsize > PAMax() then outputsize = PAMax();

if outputsize < 48 && !IsZero(baseregister<47:outputsize>) then
  level = 0;
  result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
  return result;

// Bottom bound of the Base address is:
// Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
// (Size of Input Address)/(Address per level)^(Num levels remaining)*(Size of Table))
// Log2(Num of entries*8)
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
if outputsize == 52 then
  z = (if baselowerbound < 6 then 6 else baselowerbound);
  baseaddress = baseregister<5:2>:baseregister<47:z>:Zeros(z);
else
  baseaddress = ZeroExtend(baseregister<47:baselowerbound>:Zeros(baselowerbound));

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

apply_nvnv1_effect = HaveNVExt() && EL2Enabled() && HCR_EL2.<NV,NV1> == '11' && S1TranslationRegime()

repeat
  addrselectbottom = (3-level)*stride + grainsize;
  bits(52) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
  descaddr.paddress.address = baseaddress OR index;
  descaddr.paddress.NS = ns_table;
  // If there are two stages of translation, then the first stage table walk addresses
  // are themselves subject to translation
  if secondstage || !HasS2Translation() || (HaveNV2Ext() && acctype == AccType_NV2REGISTER) then
    descaddr2 = descaddr;
  else
    hwupdatewalk = FALSE;
    descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
    // Check for a fault on the stage 2 walk
    if IsFault(descaddr2) then
      result.addrdesc.fault = descaddr2.fault;
      return result;
  
  // Update virtual address for abort functions
  descaddr2.vaddress = ZeroExtend(vaddress);
  accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
  desc = _Mem[descaddr2, 8, accdesc];
  if reversedescriptors then desc = BigEndianReverse(desc);

if desc<0> == '0' || (desc<1:0> == '01' && (level == 3 ||
  HaveBlockBBM() && IsBlockDescriptorNTBitValid()) &&
  // Fault (00), Reserved (10), Block (01) at level 3, or Block(01) with nT bit set.
  result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
return result;

// Valid Block, Page, or Table entry
if desc<1:0> == '01' || level == 3 then // Block (01) or Page (11)
    blocktranslate = TRUE;
else // Table (11)
    if (outputsize < 52 && largegrain && !IsZero(desc<15:12>)) || (outputsize < 48 && !IsZero(desc<47:outputsize>))
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
        return result;
if outputsize == 52 then
    baseaddress = desc<15:12>:desc<47:grainsize>:Zeros(grainsize);
else
    baseaddress = ZeroExtend(desc<47:grainsize>:Zeros(grainsize));
if !secondstage then // Unpack the upper and lower table attributes
    ns_table = ns_table OR desc<63>;
if !secondstage && !hierattrsdisabled then
    ap_table<1> = ap_table<1> OR desc<62>; // read-only
    if apply_nvnv1_effect then
        pxn_table = pxn_table OR desc<60>;
    else
        xn_table = xn_table OR desc<60>;
    // pxn_table and ap_table[0] apply in EL1&0 or EL2&0 translation regimes
    if !singlepriv then
        if !apply_nvnv1_effect then
            pxn_table = pxn_table OR desc<59>;
        else
            ap_table<0> = ap_table<0> OR desc<61>; // privileged
        end if
    end if
    level = level + 1;
    addrselecttop = addrselectbottom - 1;
    blocktranslate = FALSE;
end until blocktranslate;

// Check block size is supported at this level
if level < firstblocklevel then
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;

// Check for misprogramming of the contiguous bit
if largegrain then
    contiguousbitcheck = level == 2 && inputsize < 34;
elseif midgrain then
    contiguousbitcheck = level == 2 && inputsize < 30;
else
    contiguousbitcheck = level == 1 && inputsize < 34;
if contiguousbitcheck && desc<52> == '1' then
    if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
        return result;
    end if
end if

// Check the output address is inside the supported range
if (outputsize < 52 && largegrain || IsZero(desc<15:12>)) || (outputsize < 48 && !IsZero(desc<47:outputsize>))
    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;

// Unpack the descriptor into address and upper and lower block attributes
if outputsize == 52 then
    outputaddress = desc<15:12>:desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
else
    outputaddress = ZeroExtend(desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>);
// Check Access Flag
if desc<10> == '0' then
    if !update_AF then

end if
end if
end if
end if
end if
end if
result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress, sl_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);

return result;

else
result.descupdate.AF = TRUE;

if update_AP && desc<51> == '1' then
  // If hw update of access permission field is configured consider AP[2] as '0' / S2AP[2] as '1'
  if !secondstage && desc<7> == '1' then
    desc<7> = '0';
    result.descupdate.AP = TRUE;
  elsif secondstage && desc<7> == '0' then
    desc<7> = '1';
    result.descupdate.AP = TRUE;

  // Required descriptor if AF or AP[2]/S2AP[2] needs update
  result.descupdate.descaddr = descaddr;
  if apply_nvnr_effect then
    pxn = desc<54>;                                       // Bit[54] of the block/page descriptor holds PXN instead of UXN
    xn = '0';                                             // XN is '0'
  else
    xn = desc<54>;                                        // Bit[54] of the block/page descriptor holds UXN
    pxn = desc<53>;                                       // Bit[53] of the block/page descriptor holds PXN
  contiguousbit = desc<52>;
  nG = desc<11>;
  sh = desc<9:8>;
  memattr = desc<5:2>;                                   // Required descriptor if AF needs update
  if apply_nvnv1_effect then
    pxn = desc<54>;                                       // Bit[54] of the block/page descriptor holds PXN instead of UXN
    xn = '0';                                             // XN is '0'
  else
    xn = desc<54>;                                        // Bit[54] of the block/page descriptor holds UXN
    pxn = desc<53>;                                       // Bit[53] of the block/page descriptor holds PXN

  // Stage 1 block or pages might be guarded
  result.GP = desc<50>;
  result.addrdesc.memattrs = AArch64.S1AttrDecode(sh, memattr<2:0>, acctype);
  result.addrdesc.paddress.NS = memattr<3> OR ns_table;
else
result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_PTW);

  result.addrdesc.paddress.NS = nsaccess;
result.addrdesc.paddress.address = outputaddress;
result.addrdesc.fault = AArch64.NoFault();
result.contiguous = contiguousbit == '1';
if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;
return result;

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```c
// ClearStickyErrors()
// -------------------
ClearStickyErrors()
EDSCR.TXU = '0';            // Clear TX underrun flag
EDSCR.RXO = '0';            // Clear RX overrun flag
if Halted() then            // in Debug state
    EDSCR.ITO = '0';        // Clear ITR overrun flag

// If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
// The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
// in the pseudocode.
if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
    return;
EDSCR.ERR = '0';            // Clear cumulative error flag
return;
```

Library pseudocode for shared/debug/DebugTarget/DebugTarget

```c
// DebugTarget()
// -----------
// Returns the debug exception target Exception level

bits(2) DebugTarget()
secure = IsSecure();
return DebugTargetFrom(secure);

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```c
// DebugTargetFrom()
// ---------------

bits(2) DebugTargetFrom(boolean secure)
if HaveEL(EL2) && !secure then
    if ELUsingAArch32(EL2) then
        route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
    else
        route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;
if route_to_el2 then
    target = EL2;
elseif HaveEL(EL3) && HighestELUsingAArch32() && secure then
    target = EL3;
else
    target = EL1;
return target;
```
Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

// DoubleLockStatus()
// ===============
// Returns the state of the OS Double Lock.
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
if !HaveDoubleLock() then
    return FALSE;
elif ELUsingAArch32(EL1) then
    return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
else
    return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();

Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

// AllowExternalDebugAccess()
// =========================
// Returns TRUE if the External Debugger access is allowed.

boolean AllowExternalDebugAccess(boolean access_is_secure)
// The access may also be subject to OS lock, power-down, etc.
if HaveSecureExtDebugView() || ExternalDebugEnabled() then
    if HaveSecureExtDebugView() then
        allow_secure = access_is_secure;
    else
        allow_secure = ExternalSecureDebugEnabled();
    if allow_secure then
        return TRUE;
elif HaveEL(EL3) then
    return (if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD) == '0';
else
    return !IsSecure();
else
    return FALSE;

Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

// AllowExternalPMUAccess()
// ========================
// Returns TRUE if the External Debugger access is allowed.

boolean AllowExternalPMUAccess(boolean access_is_secure)
// The access may also be subject to OS lock, power-down, etc.
if HaveSecureExtDebugView() || ExternalNoninvasiveDebugEnabled() then
    if HaveSecureExtDebugView() then
        allow_secure = access_is_secure;
    else
        allow_secure = ExternalSecureNoninvasiveDebugEnabled();
    if allow_secure then
        return TRUE;
elif HaveEL(EL3) then
    return (if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD) == '0';
else
    return !IsSecure();
else
    return FALSE;

Library pseudocode for shared/debug/authentication/Debug_authentication

signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
// ExternalDebugEnabled()
// -----------------------

boolean ExternalDebugEnabled()
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.
return DBGEN == HIGH;

// ExternalHypInvasiveDebugEnabled()
// --------------------------------

boolean ExternalHypInvasiveDebugEnabled()
// In the recommended interface, ExternalHypInvasiveDebugEnabled returns the state of the (DBGEN AND HIDEN) signal.
return ExternalDebugEnabled() && HIDEN == HIGH;

// ExternalHypNoninvasiveDebugEnabled()
// ----------------------------------

boolean ExternalHypNoninvasiveDebugEnabled()
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, ExternalHypNoninvasiveDebugEnabled returns the state of the (DBGEN OR NIDEN) AND (HIDEN OR HNIDEN) signal.
return ExternalNoninvasiveDebugEnabled() && (HIDEN == HIGH || HNIDEN == HIGH);

// ExternalNoninvasiveDebugAllowed()
// ---------------------------------

boolean ExternalNoninvasiveDebugAllowed()
// Return TRUE if Trace and PC Sample-based Profiling are allowed
return (ExternalNoninvasiveDebugEnabled() && (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled() || (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUNIDEN == '1')));

// ExternalNoninvasiveDebugEnabled()
// --------------------------------

boolean ExternalNoninvasiveDebugEnabled()
// This function returns TRUE if the v8.4 Debug relaxations are implemented, otherwise this function is IMPLEMENTATION DEFINED.
// In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN OR NIDEN) signal.
return !HaveNoninvasiveDebugAuth() || ExternalDebugEnabled() || NIDEN == HIGH;

// ExternalSecureDebugEnabled()
// ----------------------------

boolean ExternalSecureDebugEnabled()
if !HaveEL(EL3) && !IsSecure() then return FALSE;
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.
return ExternalDebugEnabled() && SPIDEN == HIGH;
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
boolean ExternalSecureNoninvasiveDebugEnabled()
if !HaveEL(EL3) && !IsSecure() then return FALSE;
// If the v8.4 Debug relaxations are implemented, this function returns the value of
// ExternalSecureDebugEnabled(). Otherwise the definition of this function is
// IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.
if HaveNoninvasiveDebugAuth() then
    return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
else
    return ExternalSecureDebugEnabled();

Library pseudocode for shared/debug/cti/CTI_SetEventLevel

// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(CrossTriggerIn id, signal level);

Library pseudocode for shared/debug/cti/CTI_SignalEvent

// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);

Library pseudocode for shared/debug/cti/CrossTrigger

genereation CrossTriggerOut
    CrossTriggerOut_DebugRequest,
    CrossTriggerOut_RestartRequest,
    CrossTriggerOut_IRQ,
    CrossTriggerOut_RSVD3,
    CrossTriggerOut_TraceExtIn0,
    CrossTriggerOut_TraceExtIn1,
    CrossTriggerOut_TraceExtIn2,
    CrossTriggerOut_TraceExtIn3;

genereation CrossTriggerIn
    CrossTriggerIn_CrossHalt,
    CrossTriggerIn_PMUOverflow,
    CrossTriggerIn_RSVD2,
    CrossTriggerIn_TraceExtOut0,
    CrossTriggerIn_TraceExtOut1,
    CrossTriggerIn_TraceExtOut2,
    CrossTriggerIn_TraceExtOut3;

Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

// CheckForDCCInterrupts()
// -----------------------
CheckForDCCInterrupts()
commrnx = (EDSCR.RXfull == '1');
commttx = (EDSCR.TXfull == '0');

// COMMRX and COMMTX support is optional and not recommended for new designs.
// SetInterruptRequestLevel(InterruptID_COMMRX, if commrnx then HIGH else LOW);
// SetInterruptRequestLevel(InterruptID_COMMTX, if commttx then HIGH else LOW);

// The value to be driven onto the common COMMIRQ signal.
if ELUsingAArch32(EL1) then
    commirq = ((commrnx && DBGDCINT.RX == '1') ||
        (commttx && DBGDCINT.TX == '1'));
else
    commirq = ((commrnx && MDCCINT_EL1.RX == '1') ||
        (commttx && MDCCINT_EL1.TX == '1'));
SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);
return;
Library pseudocode for shared/debug/dccanditr/DBGDTRRX_EL0

// DBGDTRRX_EL0[] (external write)
// -----------------------------------------------
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

if EDPRSR<6:5,0> != '001' then
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return;
endif

if EDSCR.ERR == '1' then return; // Error flag set: ignore write

if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write

if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
    EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
    return;
endif

EDSCR.RXfull = '1';
DTRRX = value;

if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)
endif

if !UsingAArch32() then
    ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
    ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"
    X[1] = bits(64) UNKNOWN;
else
    ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
    ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
    R[1] = bits(32) UNKNOWN;
endif

if EDSCR.ERR == '1' then
    EDSCR.RXfull = bit UNKNOWN;
    DBGDTRRX_EL0 = bits(32) UNKNOWN;
else
    "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
    assert EDSCR.RXfull == '0';
endif

EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

return;

// DBGDTRRX_EL0[] (external read)
// -----------------------------------------------

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
return DTRRX;
// DBGDTRTX_EL0[] (external read)
// ----------------------------------------------
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return bits(32) UNKNOWN;

underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
value = if underrun then bits(32) UNKNOWN else DTRTX;
if EDSCR.ERR == '1' then return value; // Error flag set: no side-effects

if memory_mapped && EDLSR.SLK == '1' then // Software lock locked: no side-effects
    return value;
if underrun then
    EDSCR.TXU = '1';  EDSCR.ERR = '1'; // Underrun condition: block side-effects
    return value; // Return UNKNOWN
    EDSCR.TXfull = '0';
if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0'; // See comments in EDITR[] (external write)

    if !UsingAArch32() then
        ExecuteA64(0xB8404401<31:0>); // A64 "LDR W1,[X0],#4"
    else
        ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "LDR R1,[R0],#4"
    // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.TXfull = bit UNKNOWN;
        DBGDTRTX_EL0 = bits(32) UNKNOWN;
    else
        if !UsingAArch32() then
            ExecuteA64(0xD5130501<31:0>); // A64 "MSR DBGDTRTX_EL0,X1"
        else
            ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MSR DBGDTRTXint,R1"
        // "MSR DBGDTRTX_EL0,X1" calls DBGDTX_ELO[] (write) which sets TXfull.
        assert EDSCR.TXfull == '1';

        if !UsingAArch32() then
            X[1] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;
        EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
        return value;

// DBGDTRTX_EL0[] (external write)
// ----------------------------------
// The Software lock is OPTIONAL.
DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
// Software lock locked: ignore write
DTRTX = value;
return;
Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

```plaintext
// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ====================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
            result<31:0> = DTRRX;
        EDSCR.RXfull = '0';
        return result;
```

Library pseudocode for shared/debug/dccanditr/DTR

```plaintext
bits(32) DTRRX;
bits(32) DTRTX;
```
// EDITR[] (external write)
// ==============
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
if EDPRSR<6:5,0> != '001' then
  IMPLEMENTATION_DEFINED "signal slave-generated error";
  return;

if EDSCR.ERR == '1' then return;  // Error flag set: ignore write

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return;  // Software lock locked: ignore write

if !Halted() then return;  // Non-debug state: ignore write

if EDSCR.ITE == '0' || EDSCR.MA == '1' then
  EDSCR.ITO = '1';  EDSCR.ERR = '1';
  // Overrun condition: block write
  return;

// ITE indicates whether the processor is ready to accept another instruction; the processor
// may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
// is no indication that the pipeline is empty (all instructions have completed). In this
// pseudocode, the assumption is that only one instruction can be executed at a time,
// meaning ITE acts like "InstrCompl".
EDSCR.ITE = '0';

if !UsingAArch32() then
  ExecuteA64(value);
else
  ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

EDSCR.ITE = '1';

return;
DCPSInstruction(bits(2) target_el)

SynchronizeContext();

case target_el of
  when EL1
    if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then UndefinedFault();
    else handle_el = EL1;
  when EL2
    if !HaveEL(EL2) then UndefinedFault();
    elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
    elsif !IsSecureEL2Enabled() && IsSecure() then UndefinedFault();
    else handle_el = EL2;
  when EL3
    if EDSCR.SDD == '1' || !HaveEL(EL3) then UndefinedFault();
    handle_el = EL3;
  otherwise
    Unreachable();

  from_secure = IsSecure();
  if ELUsingAArch32(handle_el) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    assert UsingAArch32(); // Cannot move from AArch64 to AArch32
  case handle_el of
    when EL1
      AArch32.WriteMode(M32_Svc);
      if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    when EL2
      AArch32.WriteMode(M32_Hyp);
    when EL3
      AArch32.WriteMode(M32_Monitor);
      if HavePANExt() then
        if !from_secure then
          PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
          PSTATE.PAN = '1';
        if handle_el == EL2 then
          ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
        else
          LR = bits(32) UNKNOWN;
          SPSR[] = bits(32) UNKNOWN;
          PSTATE.E = SCTLR[].EE;
          DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
        end
        if !UsingAArch32() then
          AArch64.MaybeZeroRegisterUppers();
          MaybeZeroSVEUppers(target_el);
          PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
          if HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
            (handle_el == EL2 && HCR_EL2.E2H == '1' &&
            HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0')) then
            PSTATE.PAN = '1';
          ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(32) UNKNOWN;  ESR[] = bits(32) UNKNOWN;
          DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;
          if HaveUAOExt() then PSTATE.UAO = '0';
          UpdateEDSCRFields(); // Update EDSCR PE state flags
          sync_errors = HaveIESB() && SCTLR[].IESB == '1';
          if HaveDoubleFaultExt() && !UsingAArch32() then
            sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
            // SCTLR[].IESB might be ignored in Debug state.
            if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
              .

Shared Pseudocode Functions
sync_errors = FALSE;
if sync_errors then
    SynchronizeErrors();
return;

Library pseudocode for shared/debug/halting/DRPSInstruction

// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state
DRPSInstruction()

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
if HaveDoubleFaultExt() & !UsingAArch32() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
// SCTLR[].IESB might be ignored in Debug state.
if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
if sync_errors then
    SynchronizeErrors();

SetPSTATEFromPSR(SPSR[]);

// PSTATE.<N,Z,C,V,Q,SS,D,A,I,F> are not observable and ignored in Debug state, so
// behave as if UNKNOWN.
if UsingAArch32() then
    PSTATE.<N,Z,C,V,Q,SS,D,A,I,F> = bits(13) UNKNOWN;
// In AArch32, all instructions are T32 and unconditional.
    PSTATE.IT = '00000000'; PSTATE.T = '1';  // PSTATE.J is RES0
    DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
else
    PSTATE.<N,Z,C,V,Q,SS,D,A,I,F> = bits(9) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(32) UNKNOWN;

UpdateEDSCRFields();  // Update EDSCR PE state flags
return;

Library pseudocode for shared/debug/halting/DebugHalt

constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';

Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

    DisableITRAndResumeInstructionPrefetch();

Library pseudocode for shared/debug/halting/ExecuteA64

    // Execute an A64 instruction in Debug state.
    ExecuteA64(bits(32) instr);
Library pseudocode for shared/debug/halting/ExecuteT32

// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);

Library pseudocode for shared/debug/halting/ExitDebugState

// ExitDebugState()
// ================

ExitDebugState()
assert Halted();
SynchronizeContext();

// Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
deetect that the PE has restarted.
EDSCR.STATUS = '000001'; // Signal restarting
EDESR<2:0> = '000'; // Clear any pending Halting debug events

bits(64) new_pc;
bits(32) spsr;
if UsingAArch32() then
  new_pc = ZeroExtend(DLR);
spsr = DSPSR;
else
  new_pc = DLR_EL0;
spsr = DSPSR_EL0;
// If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0

if UsingAArch32() then
  if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
  BranchTo(new_pc<31:0>, BranchType_DBGEXIT); // AArch32 branch
else
  // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
if spsr<4> == '1' & ConstrainUnpredictableBool(Unpredictable_RESTARTZEROUPPERPC) then
  new_pc<63:32> = Zeros();
  BranchTo(new_pc, BranchType_DBGEXIT); // A type of branch that is never predicted

(EDSCR.STATUS,EDPRSR.SDR) = ('000010','1'); // Atomically signal restarted
UpdateEDSCRFields(); // Stop signalling PE state
DisableITRAndResumeInstructionPrefetch();

return;
// Halt()
// ======

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt
    if UsingAArch32() then
        DLR = ThisInstrAddr();
        DSPSR = GetPSRFromPSTATE();
        DSPSR.SS = PSTATE.SS;                           // Always save PSTATE.SS
    else
        DLR_EL0 = ThisInstrAddr();
        DSPSR_EL0 = GetPSRFromPSTATE();
        DSPSR_EL0.SS = PSTATE.SS;                      // Always save PSTATE.SS
    EDSCR.ITE = '1';   EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                           // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureDebugEnabled() then '0' else '1';
    else
        assert EDSCR.SDD == '1';                    // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';
    // PSTATE.<SS,D,A,I,F> are not observable and ignored in Debug state, so behave as if
    // UNKNOWN. PSTATE.<N,Z,C,V,Q,GE> are also not observable, but since these are not changed on
    // exception entry, this function also leaves them unchanged. PSTATE.<E,M,nRW,EL,SP> are
    // unchanged. PSTATE.IL is set to 0.
    if UsingAArch32() then
        PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    // In AArch32, all instructions are T32 and unconditional.
    // PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
    else
        PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                      // Signal entered Debug state
    UpdateEDSCRFields();                       // Update EDSCR PE state flags.

return;

// HaltOnBreakpointOrWatchpoint()
// -----------------------------
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';

// Halted()
// -------

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});  // Halted
Library pseudocode for shared/debug/halting/HaltingAllowed

```java
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elsif IsSecure() then
        return ExternalSecureDebugEnabled();
    else
        return ExternalDebugEnabled();
```

Library pseudocode for shared/debug/halting/Restarting

```java
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';  // Restarting
```

Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```java
StopInstructionPrefetchAndEnableITR();
```

Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```java
// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

UpdateEDSCRFields()
    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;
        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = if IsSecure() then '0' else '1';
        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
        else
            RW<0> = if UsingAAArch32() then '0' else '1';
            if !HaveEL(EL2) || (HaveEL(EL3) & SCR_GEN[].NS == '0' & !IsSecureEL2Enabled()) then
                RW<2> = RW<1>;
            else
                RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
                if !HaveEL(EL3) then
                    RW<3> = RW<2>;
                else
                    RW<3> = if ELUsingAAArch32(EL3) then '0' else '1';
            end if
        end if
    end if
    // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
    if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
    elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
    elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
    EDSCR.RW = RW;
    return;
```

Shared Pseudocode Functions
// CheckExceptionCatch()
// ===============
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
{
    // Called after an exception entry or exit, that is, such that IsSecure() and PSTATE.EL are correct
    // for the exception target.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() then
        if HaveExtendedECDebugEvents() then
            exception_exit = !exception_entry;
            ctrl = EDECCR<UInt>(PSTATE.EL) + base + 8>:EDECCR<UInt>(PSTATE.EL) + base>;
            case ctrl of
                when '00' halt = FALSE;
                when '01' halt = TRUE;
                when '10' halt = (exception_exit == TRUE);
                when '11' halt = (exception_entry == TRUE);
            else
                halt = (EDECCR<UInt>(PSTATE.EL) + base> == '1');
        if halt then Halt(DebugHalt_ExceptionCatch);
    }

// CheckHaltingStep()
// ===========
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
{
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
    }

// CheckOSUnlockCatch()
// =============
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
{
    if EDECR.OSUCE == '1' && !Halted() then EDESR.OSUC = '1';

// CheckPendingOSUnlockCatch()
// ===============
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
{
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);  

// CheckPendingResetCatch()
// ===============
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
{
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
Library pseudocode for shared/debug/haltingevents/CheckResetCatch

// CheckResetCatch()
// -------------------
// Called after reset

CheckResetCatch()
    if EDECR.RCE == '1' then
        EDES.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt (DebugHalt_ResetCatch);

Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

// CheckSoftwareAccessToDebugRegisters()
// -------------------------------------
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLRS.OLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt (DebugHalt_SoftwareAccess);

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

// ExternalDebugRequest()
// ----------------------

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt (DebugHalt_EDBGRQ);
        // Otherwise the CTI continues to assert the debug request until it is taken.

Library pseudocode for shared/debug/haltingevents/HaltingStep_DidNotStep

// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();

Library pseudocode for shared/debug/haltingevents/HaltingStep_SteppedEX

// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();
Library pseudocode for shared/debug/haltingevents/RunHaltingStep

```c
// RunHaltingStep()
// ================
RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
boolean reset)
// "exception_generated" is TRUE if the previous instruction generated a synchronous exception
// or was cancelled by an asynchronous exception.
// if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
// "syscall" is TRUE if the exception is a synchronous exception where the preferred return
// address is the instruction following that which generated the exception.
// "reset" is TRUE if exiting reset state into the highest EL.
if reset then assert !Halted(); // Cannot come out of reset halted
active = EDECR.SS == '1' && !Halted();
if active && reset then // Coming out of reset with EDECR.SS set
EDESR.SS = '1';
elsif active && HaltingAllowed() then
  if exception_generated && exception_target == EL3 then
    advance = syscall || ExternalSecureDebugEnabled();
  else
    advance = TRUE;
  if advance then EDESR.SS = '1';
return;
```

Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```c
// ExternalDebugInterruptsDisabled()
// --------------------------------
// Determine whether EDSCR disables interrupts routed to 'target'
boolean ExternalDebugInterruptsDisabled(bits(2) target)
case target of
  when EL3
    int_dis = EDSCR.INTdis == '11' && ExternalSecureDebugEnabled();
  when EL2
    int_dis = EDSCR.INTdis == '1x' && ExternalDebugEnabled();
  when EL1
    if IsSecure() then
      int_dis = EDSCR.INTdis == '1x' && ExternalSecureDebugEnabled();
    else
      int_dis = EDSCR.INTdis != '00' && ExternalDebugEnabled();
return int_dis;
```

Library pseudocode for shared/debug/interrupts/InterruptID

```c
enumeration InterruptID {
  InterruptID_PMUIRQ, InterruptID_COMMIRQ, InterruptID_CTIIIRQ,
  InterruptID_COMMRX, InterruptID_COMMTX};
```

Library pseudocode for shared/debug/interrupts/SetInterruptRequestLevel

```c
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```
CreatePCSample()

In a simple sequential execution of the program, CreatePCSample is executed each time the PE executes an instruction that can be sampled. An implementation is not constrained such that reads of EDPCSRlo return the current values of PC, etc.

pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
pc_sample.pc = ThisInstrAddr();
pc_sample.el = PSTATE.EL;
pc_sample.rw = if UsingAArch32() then '0' else '1';
pc_sample.ns = if IsSecure() then '0' else '1';
pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1;
if EL2Enabled() then
  if ELUsingAArch32(EL2) then
    pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
  elseif !Have16bitVMID() || VTCR_EL2.VS == '0' then
    pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
  else
    pc_sample.vmid = VTTBR_EL2.VMID;
  if HaveVirtHostExt() && !ELUsingAArch32(EL2) then
    pc_sample.contextidr_el2 = CONTEXTIDR_EL2;
  else
    pc_sample.contextidr_el2 = bits(32) UNKNOWN;
  pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
return;
EDPCSRlo is a library pseudocode for shared/debug/samplebasedprofiling/EDPCSRlo. The code snippet below demonstrates the implementation of EDPCSRlo[] (read). It includes a masked check on DLK, OSLK and PU bits, and handles software locking. The code also updates the EDPCSRhi and EDVIDSR based on various conditions. The pc_sample struct is defined, including valid, pc, el, rw, ns, has_el2, contextidr, contextidr_el2, el0h, and vmid fields. The type definition for PCSample is also shown.
Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

```c
// PMPCSR[] (read)
// -----------------
bits(32) PMPCSR[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error"
    return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = 'memory_mapped || PMLSR.SLK == '0'; // Software locked: no side-effects

if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
        PMPCSR.EL = pc_sample.el;
        PMPCSR.NS = pc_sample.ns;
        PMCID1SR = pc_sample.contextidr;
        PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;
        PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1, EL0} && !pc_sample.el0h
            then pc_sample.vmid else bits(16) UNKNOWN);
    else
        sample = Ones(32);
        if update then
            PMPCSR<55:32> = bits(24) UNKNOWN;
            PMPCSR.EL = bits(2) UNKNOWN;
            PMPCSR.NS = bit UNKNOWN;
            PMCID1SR = bits(32) UNKNOWN;
            PMCID2SR = bits(32) UNKNOWN;
            PMVIDSR.VMID = bits(16) UNKNOWN;
    
    return sample;
```

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```c
// CheckSoftwareStep()
// -------------------
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

// Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
// AArch32 state. However, because Software Step is only active when the debug target Exception
// level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
if !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() then
    if MDSCR_EL1.SS == '1' && PSTATE.SS == '0' then
        AArch64.SoftwareStepException();
```
// DebugExceptionReturnSS()
// ------------------------
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(32) spsr)
assert Halted() || Restarting() || PSTATE.EL != EL0;
SS_bit = '0';

if MDSCR_EL1.SS == '1' then
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();
    endif

if IllegalExceptionReturn(spsr) then
    dest = PSTATE.EL;
else
    (valid, dest) = ELFromSPSR(spsr);  assert valid;
    secure = IsSecureBelowEL3() || dest == EL3;

if ELUsingAArch32(dest) then
    enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
else
    mask = spsr<9>;
    enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);
endif

ELd = DebugTargetFrom(secure);
if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
    SS_bit = spsr<21>;
return SS_bit;

Library pseudocode for shared/debug/softwarestep/SSAdvance

// SSAdvance()
// -----------
// Advance the Software Step state machine.

SSAdvance()
// A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
// current Software Step state machine. However, this check is made to illustrate that the
// processor only needs to consider advancing the state machine from the active-not-pending
// state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';
return;

Library pseudocode for shared/debug/softwarestep/SoftwareStep_DidNotStep

// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean SoftwareStep_DidNotStep();

Library pseudocode for shared/debug/softwarestep/SoftwareStep_SteppedEX

// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean SoftwareStep_SteppedEX();
// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

bits(5) syndrome;

if UsingAArch32() then
    cond = AArch32.CurrentCond();
    if PSTATE.T == '0' then // A32
        syndrome<4> = '1';
        // A conditional A32 instruction that is known to pass its condition code check
        // can be presented either with COND set to 0xE, the value for unconditional, or
        // the COND value held in the instruction.
        if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
            syndrome<3:0> = '1110';
        else
            syndrome<3:0> = cond;
    else                              // T32
        // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
        // * CV set to 0 and COND is set to an UNKNOWN value
        // * CV set to 1 and COND is set to the condition code for the condition that
        //   applied to the instruction.
        if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
            syndrome<4> = '1';
            syndrome<3:0> = cond;
        else
            syndrome<4> = '0';
            syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

return syndrome;
Library pseudocode for shared/exceptions/exceptions/Exception

```plaintext
enumeration Exception | Exception_Uncategorized,       // Uncategorized or unknown reason
                        Exception_WFXTrap,             // Trapped WFI or WFE instruction
                        Exception_CP15RTTrap,          // Trapped AArch32 MCR or MRRC access to CP15
                        Exception_CP15RRTTrap,         // Trapped AArch32 MCRR or MRRC access to CP15
                        Exception_CP14RTTrap,          // Trapped AArch32 MCR or MRRC access to CP14
                        Exception_CP14DTTrap,          // Trapped AArch32 LDC or STC access to CP14
                        Exception_AdvSIMDFPAccessTrap, // HCPTTR-trapped access to SIMD or FP
                        Exception_FPIDTrap,            // Trapped access to SIMD or FP ID register
                        Exception_PACTrap,             // Trapped invalid PAC use
                        Exception_CP14RRTTrap,         // Trapped MRRC access to CP14 from AArch32
                        Exception_IllegalState,        // Illegal Execution state
                        Exception_SupervisorCall,      // Supervisor Call
                        Exception_HypervisorCall,      // Hypervisor Call
                        Exception_MonitorCall,         // Monitor Call or Trapped SMC instruction
                        Exception_SystemRegisterTrap,  // Trapped MRS or MSR system register access
                        Exception_ERetTrap,            // Trapped invalid ERET use
                        Exception_InstructionAbort,    // Instruction Abort or Prefetch Abort
                        Exception_PCAlignment,         // PC alignment fault
                        Exception_DataAbort,           // Data Abort
                        Exception_NV2DataAbort,        // Data abort at EL1 reported as being from EL2
                        Exception_SPAlignment,         // SP alignment fault
                        Exception_FPTrappedException,  // IEEE trapped FP exception
                        Exception_SError,              // SError interrupt
                        Exception_Breakpoint,         // (Hardware) Breakpoint
                        Exception_SoftwareStep,        // Software Step
                        Exception_Watchpoint,          // Watchpoint
                        Exception_SoftwareBreakpoint,  // Software Breakpoint Instruction
                        Exception_VectorCatch,         // AArch32 Vector Catch
                        Exception_IRQ,                 // IRQ interrupt
                        Exception_SVEAccessTrap,       // HCPTTR trapped access to SVE
                        Exception_BranchTarget,        // Branch Target Identification
                        Exception_SoftwareBreakpoint,  // Software Breakpoint Instruction
                        Exception_FIQ);                // FIQ interrupt
```

Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```plaintext
type ExceptionRecord is (Exception | Exception_Uncategorized, Exception_WFXTrap, Exception_CP15RTTrap, Exception_CP15RRTTrap, Exception_CP14RTTrap, Exception_CP14DTTrap, Exception_AdvSIMDFPAccessTrap, Exception_FPIDTrap, Exception_PACTrap, Exception_CP14RRTTrap, Exception_IllegalState, Exception_SupervisorCall, Exception_HypervisorCall, Exception_MonitorCall, Exception_SystemRegisterTrap, Exception_ERetTrap, Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort, Exception_NV2DataAbort, Exception_SPAlignment, Exception_FPTrappedException, Exception_SError, Exception_Breakpoint, Exception_SoftwareStep, Exception_Watchpoint, Exception_SoftwareBreakpoint, Exception_VectorCatch, Exception_IRQ, Exception_SVEAccessTrap, Exception_BranchTarget, Exception_SoftwareBreakpoint, Exception_FIQ); // Exception class
```

Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```plaintext
// ExceptionSyndrome()
// ----------------------
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception | Exception_Uncategorized, Exception_WFXTrap, Exception_CP15RTTrap, Exception_CP15RRTTrap, Exception_CP14RTTrap, Exception_CP14DTTrap, Exception_AdvSIMDFPAccessTrap, Exception_FPIDTrap, Exception_PACTrap, Exception_CP14RRTTrap, Exception_IllegalState, Exception_SupervisorCall, Exception_HypervisorCall, Exception_MonitorCall, Exception_SystemRegisterTrap, Exception_ERetTrap, Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort, Exception_NV2DataAbort, Exception_SPAlignment, Exception_FPTrappedException, Exception_SError, Exception_Breakpoint, Exception_SoftwareStep, Exception_Watchpoint, Exception_SoftwareBreakpoint, Exception_VectorCatch, Exception_IRQ, Exception_SVEAccessTrap, Exception_BranchTarget, Exception_SoftwareBreakpoint, Exception_FIQ) {

    ExceptionRecord r;
    r.type = type;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.vaddress = Zeros();
    r.ipaddress = '0';
    r.NS = '0';
    r.ipaddress = Zeros();

    return r;
```
ReservedValue()

ReservedValue()
if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
    AArch32.TakeUndefInstrException();
else
    AArch64.UndefinedFault();

UnallocatedEncoding()

UnallocatedEncoding()
if UsingAArch32() && AArch32.ExecutingCP10or11Instr() then
    FPEXC.DEX = '0';
if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
    AArch32.TakeUndefInstrException();
else
    AArch64.UndefinedFault();

EncodeLDFSC()

EncodeLDFSC(Fault type, integer level)
bits(6) result;
case type of
when Fault_AddressSize result = '0000':level<1:0>; assert level IN {0,1,2,3};
when Fault_AccessFlag result = '0010':level<1:0>; assert level IN {1,2,3};
when Fault_Permission result = '0011':level<1:0>; assert level IN {1,2,3};
when Fault_Translation result = '0001':level<1:0>; assert level IN {0,1,2,3};
when Fault_SyncExternal result = '010000';
when Fault_SyncExternalOnWalk result = '0101':level<1:0>; assert level IN {0,1,2,3};
when Fault_SyncParity result = '011000';
when Fault_SyncParityOnWalk result = '0111':level<1:0>; assert level IN {0,1,2,3};
when Fault_AsyncParity result = '011001';
when Fault_AsyncExternal result = '010001';
when Fault_AsyncParityOnWalk result = '100001';
when Fault_Debug result = '100010';
when Fault_TLBConflict result = '110000';
when Fault_HWUpdateAccessFlag result = '110001';
when Fault_Lockdown result = '110100'; // IMPLEMENTATION DEFINED
when Fault_Exclusive result = '110101'; // IMPLEMENTATION DEFINED
otherwise
    Unreachable();
return result;
Library pseudocode for shared/functions/aborts/IPAValid

```java
// IPAValid()
// =========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.type != Fault_None;
    if fault.s2fs1walk then
        return fault.type IN {Fault_AccessFlag, Fault_Permission, Fault_Translation, Fault_AddressSize};
    elsif fault.secondstage then
        return fault.type IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;
```

Library pseudocode for shared/functions/aborts/IsAsyncAbort

```java
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE otherwise.

boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;
    return (type IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.type);
```

Library pseudocode for shared/functions/aborts/IsDebugException

```java
// IsDebugException()
// ===============

boolean IsDebugException(FaultRecord fault)
    assert fault.type != Fault_None;
    return fault.type == Fault_Debug;

```

Library pseudocode for shared/functions/aborts/IsExternalAbort

```java
// IsExternalAbort()
// ================
// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault type)
    assert type != Fault_None;
    return (type IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk, Fault_AsyncExternal, Fault_AsyncParity});

// IsExternalAbort()
// ================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.type);
```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault type)
    assert type != Fault_None;
    return (type IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk});

// IsExternalSyncAbort()
// =====================

boolean IsExternalSync Abort(FaultRecord fault)
    return IsExternalSyncAbort(fault.type);

// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.type != Fault_None;

// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE otherwise.

boolean IsSErrorInterrupt(Fault type)
    assert type != Fault_None;
    return (type IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.type);

// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.type != Fault_None;
    return fault.secondstage;

// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.type != Fault_None;

bits(11) LSInstructionSyndrome();
Library pseudocode for shared/functions/common/ASR

```plaintext
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
assert shift >= 0;
if shift == 0 then
    result = x;
else
    (result, -) = ASR_C(x, shift);
return result;
```

Library pseudocode for shared/functions/common/ASR_C

```plaintext
// ASR_C()
// ========

(bits(N), bit) ASR_C(bits(N) x, integer shift)
assert shift > 0;
shift = if shift > N then N else shift;
extended_x = SignExtend(x, shift+N);
result = extended_x<shift+N-1:shift>;
carry_out = extended_x<shift-1>;
return (result, carry_out);
```

Library pseudocode for shared/functions/common/Abs

```plaintext
// Abs()
// ======

integer Abs(integer x)
return if x >= 0 then x else -x;

real Abs(real x)
return if x >= 0.0 then x else -x;
```

Library pseudocode for shared/functions/common/Align

```plaintext
// Align()
// ========

integer Align(integer x, integer y)
return y * (x DIV y);

bits(N) Align(bits(N) x, integer y)
return Align(UInt(x), y)<N-1:0>;
```

Library pseudocode for shared/functions/common/BitCount

```plaintext
// BitCount()
// ==========

integer BitCount(bits(N) x)
integer result = 0;
for i = 0 to N-1
    if x<i> == '1' then
        result = result + 1;
return result;
```
// CountLeadingSignBits()
// ================
integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);

// CountLeadingZeroBits()
// ================
integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);

// Elem[] - non-assignment form
// ================
bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector\<e*size+size-1 : e*size>;

// Elem[] - assignment form
// ================
Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector\<(e+1)*size-1:e*size> = value;
    return;

// Extend()
// ================
bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

// Extend()
// ================
bits(N) Extend(bits(M) x, boolean unsigned)
    return Extend(x, N, unsigned);
Library pseudocode for shared/functions/common/HighestSetBit

```plaintext
// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
for i = N-1 downto 0
  if x<i> == '1' then return i;
return -1;
```

Library pseudocode for shared/functions/common/Int

```plaintext
// Int()
// ======

integer Int(bits(N) x, boolean unsigned)
result = if unsigned then UInt(x) else SInt(x);
return result;
```

Library pseudocode for shared/functions/common/IsOnes

```plaintext
// IsOnes()
// =========

boolean IsOnes(bits(N) x)
return x == Ones(N);
```

Library pseudocode for shared/functions/common/IsZero

```plaintext
// IsZero()
// =========

boolean IsZero(bits(N) x)
return x == Zeros(N);
```

Library pseudocode for shared/functions/common/IsZeroBit

```plaintext
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
return if IsZero(x) then '1' else '0';
```

Library pseudocode for shared/functions/common/LSL

```plaintext
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
assert shift >= 0;
if shift == 0 then
  result = x;
else
  (result, -) = LSL_C(x, shift);
return result;
```
// LSL_C()
// ========
(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    shift = if shift > N then N else shift;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);

// LSR()
// =====

// LSR_C()
// ========
(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    shift = if shift > N then N else shift;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// LowestSetBit()
// =============

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// ======

// Max()
// Min()
// =====
integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====
real Min(real a, real b)
    return if a <= b then a else b;

// Ones()
// ======
bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// ======
bits(N) Ones()
    return Ones(N);

// ROR()
// =====
bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;

// ROR_C()
// ======
(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);

// Replicate()
// ===========
bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
Library pseudocode for shared/functions/common/RoundDown

```plaintext
integer RoundDown(real x);
```

Library pseudocode for shared/functions/common/RoundTowardsZero

```plaintext
// RoundTowardsZero()
// ================

integer RoundTowardsZero(real x)
return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

Library pseudocode for shared/functions/common/RoundUp

```plaintext
integer RoundUp(real x);
```

Library pseudocode for shared/functions/common/SInt

```plaintext
// SInt()
// -----

integer SInt(bits(N) x)
result = 0;
for i = 0 to N-1
  if x<i> == '1' then result = result + 2^i;
  if x<N-1> == '1' then result = result - 2^N;
return result;
```

Library pseudocode for shared/functions/common/SignExtend

```plaintext
// SignExtend()
// ===========

bits(N) SignExtend(bits(M) x, integer N)
assert N >= M;
return Replicate(x<M-1>, N-M) : x;
```

Library pseudocode for shared/functions/common/UInt

```plaintext
// UInt()
// -----

integer UInt(bits(N) x)
result = 0;
for i = 0 to N-1
  if x<i> == '1' then result = result + 2^i;
return result;
```
Library pseudocode for shared/functions/common/ZeroExtend

```c
// ZeroExtend()
// ============
bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

Library pseudocode for shared/functions/common/Zeros

```c
// Zeros()
// =======
bits(N) Zeros(integer N)
    return Replicate('0',N);
// Zeros()
// =======

bits(N) Zeros()
    return Zeros(N);
```

Library pseudocode for shared/functions/crc/BitReverse

```c
// BitReverse()
// ============
bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<N-i-1> = data<i>;
    return result;
```

Library pseudocode for shared/functions/crc/HaveCRCExt

```c
// HaveCRCExt()
// ============
boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

Library pseudocode for shared/functions/crc/Poly32Mod2

```c
// Poly32Mod2()
// ============
// Poly32Mod2 on a bitstring does a polynomial Modulus over \{0,1\} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
    assert N > 32;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

Library pseudocode for shared/functions/crypto/AESInvMixColumns

```c
bits(128) AESInvMixColumns(bits (128) op);
```
Library pseudocode for shared/functions/crypto/AESInvShiftRows

```plaintext
bits(128) AESInvShiftRows(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESInvSubBytes

```plaintext
bits(128) AESInvSubBytes(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESMixColumns

```plaintext
bits(128) AESMixColumns(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESShiftRows

```plaintext
bits(128) AESShiftRows(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESSubBytes

```plaintext
bits(128) AESSubBytes(bits(128) op);
```

Library pseudocode for shared/functions/crypto/HaveAESExt

```plaintext
// HaveAESExt()
// ==============
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

```plaintext
// HaveBit128PMULLExt()
// ====================
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA1Ext

```plaintext
// HaveSHA1Ext()
// =============
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA256Ext

```plaintext
// HaveSHA256Ext()
// ===============
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```
Library pseudocode for shared/functions/crypto/HaveSHA3Ext

```plaintext
// HaveSHA3Ext()
// =============
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
   if !HasArchVersion(ARMv8p2) || !HaveSHA1Ext() && HaveSHA256Ext() then
      return false;
   return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA512Ext

```plaintext
// HaveSHA512Ext()
// ===============
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
   if !HasArchVersion(ARMv8p2) || !HaveSHA1Ext() && HaveSHA256Ext() then
      return false;
   return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSM3Ext

```plaintext
// HaveSM3Ext()
// ============
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
   if !HasArchVersion(ARMv8p2) then
      return false;
   return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSM4Ext

```plaintext
// HaveSM4Ext()
// ============
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
   if !HasArchVersion(ARMv8p2) then
      return false;
   return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/ROL

```plaintext
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
   assert shift >= 0 && shift <= N;
   if (shift == 0) then
      return x;
   return ROR(x, N-shift);
```
Library pseudocode for shared/functions/crypto/SHA256hash

```c
// SHA256hash()
// ============
bits(128) SHA256hash(bits(128) X, bits(128) Y, bits(128) W, boolean part1)
bits(32) chs, maj, t;
for e = 0 to 3
  chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
  maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
  t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
  X<127:96> = t + X<127:96>;
  Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
<Y, X> = ROL(Y : X, 32);
return (if part1 then X else Y);
```

Library pseudocode for shared/functions/crypto/SHAchoose

```c
// SHAchoose()
// ===========
bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
return (((y EOR z) AND x) EOR z);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```c
// SHAhashSIGMA0()
// ===============
bits(32) SHAhashSIGMA0(bits(32) x)
return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```c
// SHAhashSIGMA1()
// ===============
bits(32) SHAhashSIGMA1(bits(32) x)
return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

Library pseudocode for shared/functions/crypto/SHAmajority

```c
// SHAmajority()
// =============
bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
return ((x AND y) OR ((x OR y) AND z));
```

Library pseudocode for shared/functions/crypto/SHAparity

```c
// SHAparity()
// ===========
bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
return (x EOR y EOR z);
```
Library pseudocode for shared/functions/crypto/Sbox

// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
bits(8) sboxout;
bits(2048) sboxstring = ... f11d95c411f105ad80ac13188a5cd7bbd2d74d012b8e5b4b08969974a0c96777e65b9f109c56ec68418f07dec3adc4d2079ee5f3ed7cb3948<2047:0>;
sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
return sboxout;

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);

Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

// ClearExclusiveMonitors()
// ========================
// Clear the local Exclusives monitor for the executing PE.
ClearExclusiveMonitors()
ClearExclusiveLocal(ProcessorID());

Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();

Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

// Record the physical address region of size bytes starting at paddress in // the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/ProcessorID

// Return the ID of the currently executing PE.
integer ProcessorID();

Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

// AArch32.HaveHPDExt()
// -------------------
boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);

Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

// AArch64.HaveHPDExt()
// -------------------
boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);

Library pseudocode for shared/functions/extension/Have52BitPAExt

// Have52BitPAExt()
// ---------------
boolean Have52BitPAExt()
    return HasArchVersion(ARMv8p2);

Library pseudocode for shared/functions/extension/Have52BitVAExt

// Have52BitVAExt()
// ---------------
boolean Have52BitVAExt()
    return HasArchVersion(ARMv8p2);

Library pseudocode for shared/functions/extension/HaveAtomicExt

// HaveAtomicExt()
// ---------------
boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);

Library pseudocode for shared/functions/extension/HaveBTIExt

// HaveBTIExt()
// ------------
// Returns TRUE if BTI implemented and FALSE otherwise
boolean HaveBTIExt()
    return HasArchVersion(ARMv8p5);
// HaveBlockBBM()
// ==============
// Returns TRUE if support for changing block size without requiring break-before-make is implemented.

boolean HaveBlockBBM()
    return HasArchVersion(ARMv8p4);

// HaveCommonNotPrivateTransExt()
// ==============================

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);

// HaveDITExt()
// ============

boolean HaveDITExt()
    return HasArchVersion(ARMv8p4);

// HaveDOTPExt()
// =============

boolean HaveDOTPExt()
    return HasArchVersion(ARMv8p4) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has Dot Product extension");

// HaveDoubleFaultExt()
// ====================

boolean HaveDoubleFaultExt()
    return HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB();

// HaveDoubleLock()
// ================

boolean HaveDoubleLock()
    return !HasArchVersion(ARMv8p4) || boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";

// HaveE0PDExt()
// =============

boolean HaveE0PDExt()
    return HasArchVersion(ARMv8p5);
Library pseudocode for shared/functions/extension/HaveExtendedCacheSets

```java
// HaveExtendedCacheSets()
// =======================

boolean HaveExtendedCacheSets()
return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveExtendedECDebugEvents

```java
// HaveExtendedECDebugEvents()
// ===========================

boolean HaveExtendedECDebugEvents()
return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExt

```java
// HaveExtendedExecuteNeverExt()
// =============================

boolean HaveExtendedExecuteNeverExt()
return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveFCADDExt

```java
// HaveFCADDExt()
// ==============

boolean HaveFCADDExt()
return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveFJCVTZSExt

```java
// HaveFJCVTZSExt()
// ================

boolean HaveFJCVTZSExt()
return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```java
// HaveFP16MulNoRoundingToFP32Ext()
// =================================

// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate to FP32 instructions, // and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
if !HaveFP16Ext() then return FALSE;
if HasArchVersion(ARMv8p4) then return TRUE;
return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension";
```

Library pseudocode for shared/functions/extension/HaveFlagFormatExt

```java
// HaveFlagFormatExt()
// ===================

// Returns TRUE if flag format conversion instructions implemented // and FALSE otherwise

boolean HaveFlagFormatExt()
return HasArchVersion(ARMv8p5);
```
Library pseudocode for shared/functions/extension/HaveFlagManipulateExt

```cpp
// HaveFlagManipulateExt()
// =======================
// Returns TRUE if has flag manipulate instructions, and FALSE otherwise

boolean HaveFlagManipulateExt()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveFrintExt

```cpp
// HaveFrintExt()
// ==============
// Returns TRUE if FRINT instructions are implemented and FALSE otherwise

boolean HaveFrintExt()
    return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveHPMDExt

```cpp
// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveIESB

```cpp
// HaveIESB()
// ===========

boolean HaveIESB()
    return (HaveRASExt() &&
            boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
```

Library pseudocode for shared/functions/extension/HaveMPAMExt

```cpp
// HaveMPAMExt()
// ==============
// Returns TRUE if MPAM implemented and FALSE otherwise.

boolean HaveMPAMExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has MPAM extension");
```

Library pseudocode for shared/functions/extension/HaveMTEExt

```cpp
// HaveMTEExt()
// =============
// Returns TRUE if MTE implemented and FALSE otherwise.

boolean HaveMTEExt()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension";
```

Library pseudocode for shared/functions/extension/HaveNV2Ext

```cpp
// HaveNV2Ext()
// =============

boolean HaveNV2Ext()
    return HasArchVersion(ARMv8p4);
```
Library pseudocode for shared/functions/extension/HaveNVExt

```java
// HaveNVExt()
// ===========
boolean HaveNVExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveNoSecurePMUDisableOverride

```java
// HaveNoSecurePMUDisableOverride()
// ================================
boolean HaveNoSecurePMUDisableOverride()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveNoninvasiveDebugAuth

```java
// HaveNoninvasiveDebugAuth()
// ==========================
// Returns FALSE if support for the removal of the non-invasive Debug controls is implemented.
boolean HaveNoninvasiveDebugAuth()
    return !HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HavePANExt

```java
// HavePANExt()
// ============
boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```java
// HavePageBasedHardwareAttributes()
// =================================
boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HavePrivATExt

```java
// HavePrivATExt()
// ===============
boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

```java
// HaveQRDMLAHExt()
// ================
boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```
// HaveRASExt()
// ============
boolean HaveRASExt()
  return (HasArchVersion(ARMv8p2) ||
  boolean IMPLEMENTATION_DEFINED "Has RAS extension");

// HaveSBExt()
// ===========
// Returns TRUE if has SB feature support, and FALSE otherwise.
bool HaveSBExt()
  return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";

// HaveSSBSExt()
// =============
// Returns TRUE if has SSBS feature support, and FALSE otherwise.
bool HaveSSBSExt()
  return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";

// HaveSecureEL2Ext()
// ==================
// Returns TRUE if has Secure EL2, and FALSE otherwise.
bool HaveSecureEL2Ext()
  return HasArchVersion(ARMv8p4);

// HaveSecureExtDebugView()
// ========================
// Returns TRUE if supports Secure and Non-secure views of debug peripherals is implemented.
bool HaveSecureExtDebugView()
  return HasArchVersion(ARMv8p4);

// HaveSelfHostedTrace()
// =====================
// Returns TRUE if has Secure and Non-secure views of debug peripherals is implemented.
bool HaveSelfHostedTrace()
  return HasArchVersion(ARMv8p4);

// HaveSmallPageTblExt()
// =====================
// Returns TRUE if has Small Page Table Support, and FALSE otherwise
bool HaveSmallPageTblExt()
  return HasArchVersion(ARMv8p4) && boolean IMPLEMENTATION_DEFINED "Has Small Page Table extension";
// HaveStage2MemAttrControl()
// -------------------------
// Returns TRUE if support for Stage2 control of memory types and cacheability attributes is implemented.

boolean HaveStage2MemAttrControl()
    return HasArchVersion(ARMv8p4);

// HaveStatisticalProfiling()
// --------------------------

boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);

// HaveTrapLoadStoreMultipleDeviceExt()
// -------------------------------------

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);

// HaveUA16Ext()
// -------------
// Returns TRUE if has extended unaligned memory access support, and FALSE otherwise

boolean HaveUA16Ext()
    return HasArchVersion(ARMv8p4);

// HaveUAOExt()
// -----------

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);

// HaveVirtHostExt()
// ----------------

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);

// InsertIESBBeforeException()
// -----------------------------
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable SError interrupt must be taken before executing any instructions in the exception handler. However, this can be before the branch to the exception handler is made.

boolean InsertIESBBeforeException(bits(2) el);
// FixedToFP()
// ===========
// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and Rounding.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
assert N IN {16,32,64};
assert M IN {16,32,64};
bits(N) result;
assert fbits >= 0;
assert rounding != FPRounding_ODD;

// Correct signed-ness
int_operand = Int(op, unsigned);

// Scale by fractional bits and generate a real value
real_operand = Real(int_operand) / 2.0^fbits;
if real_operand == 0.0 then
    result = FPZero('0');
else
    result = FPRound(real_operand, fpcr, rounding);
return result;

// FPAbs()
// ========

bits(N) FPAbs(bits(N) op)
assert N IN {16,32,64};
return '0' : op<N-2:0>;

// FPAdd()
// ========

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    inf1 = (type1 == FPType_Infinity); inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero); zero2 = (type2 == FPType_Zero);
    if inf1 && inf2 && sign1 == NOT(sign2) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
        result = FPInfinity('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
        result = FPInfinity('1');
    elsif zero1 && zero2 && sign1 == sign2 then
        result = FPZero(sign1);
    else
        result_value = value1 + value2;
        if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
            result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero(result_sign);
        else
            result = FPRound(result_value, fpcr, rounding);
        end;
    end;
else
    result_value = value1 + value2;
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign);
    else
        result = FPRound(result_value, fpcr, rounding);
    end;
return result;
// FPCompare()  
// ===========  
bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)  
assert N IN {16, 32, 64};  
(type1, sign1, value1) = FPUnpack(op1, fpcr);  
(type2, sign2, value2) = FPUnpack(op2, fpcr);  
if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then  
  result = '0011';  
  if type1==FPType_SNaN || type2==FPType_SNaN || signal_nans then  
    FPProcessException(FPExc_InvalidOp, fpcr);  
else  
  // All non-NaN cases can be evaluated on the values produced by FPUnpack()  
  if value1 == value2 then  
    result = '0110';  
  elsif value1 < value2 then  
    result = '1000';  
  else  // value1 > value2  
    result = '0010';  
return result;

// FPCompareEQ()  
// =============  
boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)  
assert N IN {16, 32, 64};  
(type1, sign1, value1) = FPUnpack(op1, fpcr);  
(type2, sign2, value2) = FPUnpack(op2, fpcr);  
if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then  
  result = FALSE;  
  if type1==FPType_SNaN || type2==FPType_SNaN then  
    FPProcessException(FPExc_InvalidOp, fpcr);  
else  
  // All non-NaN cases can be evaluated on the values produced by FPUnpack()  
  result = (value1 == value2);  
return result;

// FPCompareGE()  
// ===============  
boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)  
assert N IN {16, 32, 64};  
(type1, sign1, value1) = FPUnpack(op1, fpcr);  
(type2, sign2, value2) = FPUnpack(op2, fpcr);  
if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then  
  result = FALSE;  
  FPProcessException(FPExc_InvalidOp, fpcr);  
else  
  // All non-NaN cases can be evaluated on the values produced by FPUnpack()  
  result = (value1 >= value2);  
return result;
Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
    result = FALSE;
    FPProcessException(FPExc_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 > value2);
return result;

Library pseudocode for shared/functions/float/fpconvert/FPConvert

// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by Rounding.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)
assert M IN {16,32,64};
assert N IN {16,32,64};
bits(M) result;

// Unpack floating-point operand optionally with flush-to-zero.
(type,sign,value) = FPUnpackCV(op, fpcr);

alt_hp = (M == 16) && (fpcr.AHP == '1');
if type == FPType_SNaN || type == FPType_QNaN then
    if alt_hp then
        result = FPZero(sign);
    elsif fpcr.DN == '1' then
        result = FPDefaultNaN();
    else
        result = FPConvertNaN(op);
    if type == FPType_SNaN || alt_hp then
        FPProcessException(FPExc_InvalidOp, fpcr);
else
    if type == FPType_Infinity then
        if alt_hp then
            result = sign:
                   OnesModule(M-1);
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPInfinity(sign);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCV(value, fpcr, rounding);
    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr)
return FPConvert(op, fpcr, FPRoundingMode(fpcr));
Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)
assert N IN {16,32,64};
assert M IN {16,32,64};
bits(M) result;
bits(51) frac;

sign = op<N-1>;

// Unpack payload from input NaN
case N of
when 64 frac = op<50:0>;
when 32 frac = op<21:0>: Zeros(29);
when 16 frac = op<8:0>: Zeros(42);

// Repack payload into output NaN, while
// converting an SNaN to a QNaN.
case M of
when 64 result = sign: Ones(M-52):frac;
when 32 result = sign: Ones(M-23):frac<50:29>;
when 16 result = sign: Ones(M-10):frac<50:42>;

return result;

Library pseudocode for shared/functions/float/fpcrtype/FPCRTypetype FPCRType;

Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

// FPDecodeRM()
// =============
// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
case rm of
when '00' return FPRounding_TIEAWAY; // A
when '01' return FPRounding_TIEEVEN; // N
when '10' return FPRounding_POSINF; // P
when '11' return FPRounding_NEGINF; // M

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

// FPDecodeRounding()
// ===============
// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
case rmode of
when '00' return FPRounding_TIEEVEN; // N
when '01' return FPRounding_POSINF; // P
when '10' return FPRounding_NEGINF; // M
when '11' return FPRounding_ZERO; // Z
Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```c
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN()
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
sign = '0';
exp = Ones(E);
frac = '1':Zeros(F-1);
return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpdiv/FPDiv

```c
// FPDiv()
// ========

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);
  if (inf1 && inf2) || (zero1 && zero2) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif inf1 || zero2 then
    result = FPInfinity(sign1 EOR sign2);
    if !inf1 then
      FPProcessException(FPExc_DivideByZero, fpcr);
    elsif zero1 || inf2 then
      result = FPZero(sign1 EOR sign2);
    else
      result = FPRound(value1/value2, fpcr);
  return result;
```

Library pseudocode for shared/functions/float/fpexc/FPExc

```c
enumeration FPExc       {
  FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
  FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```c
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = Ones(E);
frac = Zeros(F);
return sign : exp : frac;
```
Library pseudocode for shared/functions/float/fpmax/FPMax

// FPMax()
// =========

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  if value1 > value2 then
    (type,sign,value) = (type1,sign1,value1);
  else
    (type,sign,value) = (type2,sign2,value2);
  if type == FPType_Infinity then
    result = FPInfinity(sign);
  elsif type == FPType_Zero then
    sign = sign1 AND sign2; // Use most positive sign
    result = FPZero(sign);
  else
    // The use of FPRound() covers the case where there is a trapped underflow exception
    // for a denormalized number even though the result is exact.
    result = FPRound(value, fpcr);
  return result;

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

// FPMaxNormal()
// ==============

bits(N) FPMaxNormal(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = Ones(E-1):'0';
frac = Ones(F);
return sign : exp : frac;

Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

// FPMaxNum()
// ===========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);
// treat a single quiet-NaN as -Infinity
if type1 == FPType_QNaN && type2 != FPType_QNaN then
  op1 = FPInfinity('1');
elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
  op2 = FPInfinity('1');
return FPMax(op1, op2, fpcr);
Library pseudocode for shared/functions/float/fpmin/FPMin

```plaintext
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  if value1 < value2 then
    (type,sign,value) = (type1,sign1,value1);
  else
    (type,sign,value) = (type2,sign2,value2);
  if type == FPType_Infinity then
    result = FPInfinity(sign);
  elsif type == FPType_Zero then
    sign = sign1 OR sign2; // Use most negative sign
    result = FPZero(sign);
  else
    // The use of FPRound() covers the case where there is a trapped underflow exception
    // for a denormalized number even though the result is exact.
    result = FPRound(value, fpcr);
  end
end
return result;
```

Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```plaintext
// FPMinNum()
// =

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

// Treat a single quiet-NaN as +Infinity
if type1 == FPType_QNaN && type2 == FPType_QNaN then
  op1 = FPInfinity('0');
elsif type1 == FPType_QNaN && type2 == FPType_QNaN then
  op2 = FPInfinity('0');
return FPMin(op1, op2, fpcr);
```
Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// ========

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);
    if (inf1 && zero2) || (zero1 && inf2) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif inf1 || inf2 then
        result = FPInfinity(sign1 EOR sign2);
    elsif zero1 || zero2 then
        result = FPDZero(sign1 EOR sign2);
    else
        result = FPRound(value1*value2, fpcr);
    return result;
```
// FPMulAdd()
// =========
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
rounding = FPRoundingMode(fpcr);
(typeA,signA,valueA) = FPUnpack(addend, fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
inf1 = (type1 == FPTyp_Infinity); zero1 = (type1 == FPTyp_Zero);
inf2 = (type2 == FPTyp_Infinity); zero2 = (type2 == FPTyp_Zero);
(done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);
if typeA == FPTyp_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
  result = FPDefaultNaN();
  FPProcessException(FPExc_InvalidOp, fpcr);
if !done then
  infA = (typeA == FPTyp_Infinity);  zeroA = (typeA == FPTyp_Zero);
  signP = sign1 EOR sign2;
  infP  = inf1 || inf2;
  zeroP = zero1 || zero2;
  // Determine sign and type product will have if it does not cause an Invalid
  // Operation.
  signP = sign1 EOR sign2;
  infP  = inf1 || inf2;
  zeroP = zero1 || zero2;
  // Non SNaN-generated Invalid Operation cases are multiples of zero by infinity and
  // additions of opposite-signed infinities.
  if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elseif (infA && signA == '0') || (infP && signP == '0') then
    result = FPInfinity('0');
  elseif (infA && signA == '1') || (infP && signP == '1') then
    result = FPInfinity('1');
  // Cases where the result is exactly zero and its sign is not determined by the
  // rounding mode are additions of same-signed zeros.
  elseif zeroA && zeroP && signA == signP then
    result = FPZero(signA);
  // Otherwise calculate numerical result and round it.
  else
    result_value = valueA + (value1 * value2);
    if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr);

return result;
Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

// FPMulAddH()
// ===========

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, fpcr)
assert N IN {32,64};
rounding = FPRoundingMode(fpcr);
(typeA,signA,valueA) = FPUnpack(addend, fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
infinf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
infinf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
(done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);
if typeA == FPType_QNaN && ((infinf1 && zero2) || (zero1 && infinfi2)) then
result = FPDefaultNaN();

FPProcessException(FPExc_InvalidOp, fpcr);
if !done then
infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
// Determine sign and type product will have if it does not cause an Invalid
// Operation.
signP = sign1 EOR sign2;
infP = infinfi1 || infinfi2;
zeroP = zero1 || zero2;
// Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
// additions of opposite-signed infinities.
if (infinf1 && zero2) || (zero1 && infinfi2) || (infA && infP && signA != signP) then
result = FPDefaultNaN();

FPProcessException(FPExc_InvalidOp, fpcr);
// Other cases involving infinities produce an infinity of the same sign.
elsif (infA && signA == '0') || (infP && signP == '0') then
result = FPInfinity('0');
elsif (infA && signA == '1') || (infP && signP == '1') then
result = FPInfinity('1');
// Cases where the result is exactly zero and its sign is not determined by the
// rounding mode are additions of same-signed zeros.
elsif zeroA && zeroP && signA == signP then
result = FPZero(signA);
// Otherwise calculate numerical result and round it.
else
result_value = valueA + (value1 * value2);
if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
result = FPZero(result_sign);
else
result = FPRound(result_value, fpcr);
return result;
// FPProcessNaNs3H()
// -----------------

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3, bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3, FPCRType fpcr)

assert N IN {32,64};

bits(N) result;
if type1 == FPType_SNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_SNaN then
done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
elsif type3 == FPType_SNaN then
done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
elsif type1 == FPType_QNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_QNaN then
done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
elsif type3 == FPType_QNaN then
done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
else
done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);

// FPMulX()
// -------

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};

bits(N) result;
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
inf1 = (type1 == FPType_Infinity);
inf2 = (type2 == FPType_Infinity);
zero1 = (type1 == FPType_Zero);
zero2 = (type2 == FPType_Zero);
if (inf1 && zero2) || (zero1 && inf2) then
result = FPTwo(sign1 EOR sign2);
elsif inf1 || inf2 then
result = FPInfinity(sign1 EOR sign2);
elsif zero1 || zero2 then
result = FPZero(sign1 EOR sign2);
else
result = FPRound(value1*value2, fpcr);
return result;

// FPNeg()
// -------

bits(N) FPNeg(bits(N) op)

assert N IN {16,32,64};

return NOT(op<N-1>) : op<N-2:0>();
Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```plaintext
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign)
assert N IN {16,32,64};
count integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
count integer F = N - (E + 1);
exp = '0':Ones(E-1);
frac = '1':Zeros(F-1);
return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpprocessexception/FPProcessException

```plaintext
// FPProcessException()
// ================

// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr) // Determine the cumulative exception bit number
case exception of
  when FPExc_InvalidOp   cumul = 0;
  when FPExc_DivideByZero cumul = 1;
  when FPExc_Overflow     cumul = 2;
  when FPExc_Underflow    cumul = 3;
  when FPExc_Inexact     cumul = 4;
  when FPExc_InputDenorm cumul = 7;
endcase
enable = cumul + 8;
if fpcr<enable> == '1' then
  // Trapping of the exception enabled.
  // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
  // if so then how exceptions may be accumulated before calling FPTrapException()
  IMPLEMENTATION_DEFINED "floating-point trap handling"
elsif UsingAArch32() then
  // Set the cumulative exception bit
  FPSCR<cumul> = '1';
else
  // Set the cumulative exception bit
  FPSR<cumul> = '1';
return;
```

Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```plaintext
// FPProcessNaN()
// ===============

bits(N) FPProcessNaN(FPType type, bits(N) op, FPCRType fpcr)
assert N IN {16,32,64};
assert type IN {FPType_QNaN, FPType_SNaN};

case N of
  when 16 topfrac = 9;
  when 32 topfrac = 22;
  when 64 topfrac = 51;
endcase
result = op;
if type == FPType_SNaN then
  result<topfrac> = '1';
  FPProcessException(FPExc_InvalidOp, fpcr);
if fpcr.DN == '1' then  // DefaultNaN requested
  result = FPDefaultNaN();
return result;
```
// FPProcessNaNs()
// ===============
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
if type1 == FPType_SNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_SNaN then
done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elsif type1 == FPType_QNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_QNaN then
done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
else
done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);

// FPProcessNaNs3()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3, bits(N) op1, bits(N) op2, bits(N) op3, FPCRType fpcr)
assert N IN {16,32,64};
if type1 == FPType_SNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_SNaN then
done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elsif type3 == FPType_SNaN then
done = TRUE; result = FPProcessNaN(type3, op3, fpcr);
elsif type1 == FPType_QNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_QNaN then
done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elsif type3 == FPType_QNaN then
done = TRUE; result = FPProcessNaN(type3, op3, fpcr);
else
done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);
// FPRecipEstimate()
// -----------------

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)
assert N IN {16,32,64};
(type,sign,value) = FPUnpack(operand, fpcr);
if type == FPType_SNaN || type == FPType_QNaN then
    result = FPProcessNaN(type, operand, fpcr);
elsif type == FPType_Infinity then
    result = FPZero(sign);
elsif type == FPType_Zero then
    result = FPInfinity(sign);
    FPProcessException(FPExc_DivideByZero, fpcr);
elsif (%
    (N == 16 && Abs(value) < 2.0^-16) ||
    (N == 32 && Abs(value) < 2.0^-128) ||
    (N == 64 && Abs(value) < 2.0^-1024)
) then
    case FPRoundingMode(fpcr) of
        when FPRounding_TIEEVEN
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO
            overflow_to_inf = FALSE;
    result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
    FPProcessException(FPExc_Overflow, fpcr);
    FPProcessException(FPExc_Inexact, fpcr);
elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) && (%
    (N == 16 && Abs(value) >= 2.0^14) ||
    (N == 32 && Abs(value) >= 2.0^126) ||
    (N == 64 && Abs(value) >= 2.0^1022)
) then
    // Result flushed to zero of correct sign
    result = FPZero(sign);
    if UsingAArch32() then
        FPSR.UFC = '1';
    else
        FPSCR.UFC = '1';
    end
else
    // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
    // calculate result exponent. Scaled value has copied sign bit,
    // exponent = 1022 = double-precision biased version of -1,
    // fraction = original fraction
    case N of
        when 16
            fraction = operand<9:0> : Zeros(42);
            exp = UInt(operand<14:10>);
        when 32
            fraction = operand<22:0> : Zeros(29);
            exp = UInt(operand<30:23>);
        when 64
            fraction = operand<51:0>;
            exp = UInt(operand<62:52>);
    if exp == 0 then
        if fraction<51> == 0 then
            exp = -1;
            fraction = fraction<49:0>:'00';
        else
            fraction = fraction<50:0>:'0';
        integer scaled = UInt('1':fraction<51:44>);
    case N of
        when 16 result_exp = 29 - exp; // In range 29-30 = -1 to 29+1 = 30
        when 32 result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254

Shared Pseudocode Functions
when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

// scaled is in range 256..511 representing a fixed-point number in range [0.5..1.0)
estimate = RecipEstimate(scaled);

// estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
// Convert to scaled floating point result with copied sign bit,
// high-order bits from estimate, and exponent calculated above.

fraction = estimate<7:0> : Zeroes(44);
if result_exp == 0 then
  fraction = '1' : fraction<51:1>;
elsif result_exp == -1 then
  fraction = '01' : fraction<51:2>;
  result_exp = 0;

  case N of
    when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
    when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
    when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

  return result;

Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

// Compute estimate of reciprocal of 9-bit fixed-point number
// a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipEstimate(integer a)
  assert 256 <= a && a < 512;
  a = a*2+1; // round to nearest
  integer b = (2 ^ 19) DIV a;
  r = (b+1) DIV 2; // round to nearest
  assert 256 <= r && r < 512;
  return r;
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr)
assert N IN {16,32,64};

case N of
  when 16 esize =  5;
  when 32 esize =  8;
  when 64 esize = 11;

bits(N)           result;
bits(esize)       exp;
bits(esize)       max_exp;
bits(N-(esize+1)) frac = Zeros();

case N of
  when 16 exp = op<10+esize-1:10>;
  when 32 exp = op<23+esize-1:23>;
  when 64 exp = op<52+esize-1:52>;

max_exp = Ones(esize) - 1;

(type,sign,value) = FPUnpack(op, fpcr);
if type == FPType_SNaN || type == FPType_QNaN then
  result = FPProcessNaN(type, op, fpcr);
else
  if IsZero(exp) then // Zero and denormals
    result = sign:max_exp:frac;
  else // Infinities and normals
    result = sign:NOT(exp):frac;

return result;
Library pseudocode for shared/functions/float/fpround/FPRound
// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding)
fp.cr.AHP = '0';
return FPRoundBase(op, fpcr, rounding);

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding)
assert N IN {16,32,64};
assert op != 0.0;
assert rounding != FPRounding_TIEAWAY;
bites(N) result;

// Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
if N == 16 then
    minimum_exp = -14;  E = 5;  F = 10;
else if N == 32 then
    minimum_exp = -126;  E = 8;  F = 23;
else  // N == 64
    minimum_exp = -1022;  E = 11;  F = 52;

// Split value into sign, unrounded mantissa and exponent.
if op < 0.0 then
    sign = '1';  mantissa = -op;
else
    sign = '0';  mantissa = op;
exponent = 0;
while mantissa < 1.0 do
    mantissa = mantissa * 2.0;  exponent = exponent - 1;
while mantissa >= 2.0 do
    mantissa = mantissa / 2.0;  exponent = exponent + 1;

// Deal with flush-to-zero.
if ((fpcr.F2 == '1' && N != 16) || (fpcr.F16 == '1' && N == 16)) && exponent < minimum_exp then
    if UsingAArch32() then
        FPSCR.UFC = '1';
    else
        FPSR.UFC = '1';
    return FPZero(sign);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
bias_exp = Max(exponent - minimum_exp + 1, 0);
if bias_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if bias_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped.
if bias_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
    FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
        overflow_to_inf = TRUE;
    when FPRounding_POSINF
        round_up = (error != 0.0 && sign == '0');
        overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF
        round_up = (error != 0.0 && sign == '1');

overflow_to_inf = (sign == '1');

when FPRounding ZERO, FPRounding ODD
    round_up = FALSE;
    overflow_to_inf = FALSE;

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then // Rounded up from denormalized to normalized
        biased_exp = 1;
    if int_mant == 2^(F+1) then // Rounded up to next exponent
        biased_exp = biased_exp + 1; int_mant = int_mant DIV 2;

// Handle rounding to odd aka Von Neumann rounding
if error != 0.0 && rounding == FPRounding ODD then
    int_mant<0> = '1';

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        error = 1.0; // Ensure that an Inexact exception occurs
    else
        result = sign : biased_exp<N-F:2:0> : int_mant<F-1:0>;
else // Alternative half precision
    if biased_exp >= 2^E then
        result = sign : Ones(N-1);
        FPProcessException(FPExc_InvalidOp, fpcr);
        error = 0.0; // Ensure that an Inexact exception does not occur
    else
        result = sign : biased_exp<N-F:2:0> : int_mant<F-1:0>;

// Deal with Inexact exception.
if error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);

return result;

// FPRound()

bits(N) FPRound(real op, FPCRType fpcr)
return FPRound(op, fpcr, FPRoundingMode(fpcr));
// FPRoundingMode()
// ================
// Return the current floating-point rounding mode.
FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);

// FPRoundInt()
// ============
// Round OP to nearest integral floating point value using rounding mode Rounding.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.
bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};
    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPType_Infinity then
        result = FPInfinity(sign);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    else
        // extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);
        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
        if round_up then int_result = int_result + 1;
        // Convert integer value into an equivalent real value
        real_result = Real(int_result);
        // Re-encode as a floating-point value, result is always exact
        if real_result == 0.0 then
            result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO);
        // Generate inexact exceptions
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);
        return result;
Library pseudocode for shared/functions/float/fproundintn/FPRoundIntN
// FPRoundIntN()
// =============
bits(N) FPRoundIntN(bits(N) op, FPCRTYPE fpcr, FPRounding rounding, integer intsize)
assert rounding != FPRounding_ODD;
assert N IN (32,64);
integer exp;
constant integer E = (if N == 32 then 8 else 11);
constant integer F = N - (E + 1);

// Unpack using FPCR to determine if subnormals are flushed-to-zero
(type,sign,value) = FPUnpack(op, fpcr);

if type IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
  if N == 32 then
    exp = 126 + intsize;
    result = '1':exp<(E-1):0>:Zeros(F);
  else
    exp = 1022+intsize;
    result = '1':exp<(E-1):0>:Zeros(F);
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif type == FPType_Zero then
    result = FPZero(sign);
  else
    // Extract integer component
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case rounding of
    when FPRounding_TIEEVEN
      round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
    when FPRounding_POSINF
      round_up = error != 0.0;
    when FPRounding_NEGINF
      round_up = FALSE;
    when FPRounding_ZERO
      round_up = error != 0.0 && int_result < 0;
    when FPRounding_TIEAWAY
      round_up = error > 0.5 || (error == 0.5 && int_result >= 0);
    if round_up then int_result = int_result + 1;

    if int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1) then
      if N == 32 then
        exp = 126 + intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
      else
        exp = 1022 + intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
      // this case shouldn't set Inexact
      error = 0.0;
    else
      // Convert integer value into an equivalent real value
      real_result = Real(int_result);
      // Re-encode as a floating-point value, result is always exact
      if real_result == 0.0 then
        result = FPZero(sign);
      else
        result = FPRound(real_result, fpcr, FPRounding_ZERO);
    // Generate inexact exceptions
    if error != 0.0 then
      FPProcessException(FPExc_Inexact, fpcr);
    return result;
// FPRSqrtEstimate()
// -----------------

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPTYPE_SNAN || type == FPTYPE_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPTYPE_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif type == FPTYPE_Infinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);
        end

        if exp == 0 then
            while fraction<51> == 0 do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            end
            fraction = fraction<50:0> : '0';
        end

        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);
        end

        case N of
            when 16 result_exp = (44 - exp) DIV 2;
            when 32 result_exp = (380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;
        end

        estimate = RecipSqrtEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.

        case N of
            when 16 result = '0' : result_exp<N=12:0> : estimate<7:0> : Zeros(2);
            when 32 result = '0' : result_exp<N=25:0> : estimate<7:0> : Zeros(15);
            when 64 result = '0' : result_exp<N=54:0> : estimate<7:0> : Zeros(44);
        end

        return result;
Library pseudocode for shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

```pseudocode
// Compute estimate of reciprocal square root of 9-bit fixed-point number
// a is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipSqrtEstimate(integer a)
assert 128 <= a && a < 512;
if a < 256 then // 0.25 .. 0.5
    a = a*2+1;     // a in units of 1/512 rounded to nearest
else // 0.5 .. 1.0
    a = (a >> 1) << 1; // discard bottom bit
    a = (a+1)*2;  // a in units of 1/256 rounded to nearest
integer b = 512;
while a*(b+1)*(b+1) < 2^28 do
    b = b+1;
// b = largest b such that b < 2^14 / sqrt(a) do
r = (b+1) DIV 2; // round to nearest
assert 256 <= r && r < 512;
return r;
```

Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```pseudocode
// FPSqrt()
// --------

bits(N) FPSqrt(bits(N) op, FPCRTyp fpcr)
assert N IN {16,32,64};
(type,sign,value) = FPUnpack(op, fpcr);
if type == FPTyp_SNaN || type == FPTyp_QNaN then
    result = FPProcessNaN(type, op, fpcr);
elsif type == FPTyp_Zero then
    result = FPZero(sign);
elsif type == FPTyp_Infinity && sign == '0' then
    result = FPInfinity(sign);
elsif sign == '1' then
    result = FPDefaultNaN();
    FPProcessException(FPEx_InvalidOp, fpcr);
else
    result = FPRound(Sqrt(value), fpcr);
return result;
```
Library pseudocode for shared/functions/float/fpsub/FPSub

```plaintext
// FPSub()
// ========

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);
  if inf1 && inf2 && sign1 == sign2 then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
    result = FPInfinity('0');
  elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
    result = FPInfinity('1');
  elsif zero1 && zero2 && sign1 == NOT(sign2) then
    result = FPZero(sign1);
  else
    result_value = value1 - value2;
    if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr, rounding);
  return result;
```

Library pseudocode for shared/functions/float/fpthree/FPThree

```plaintext
// FPThree()
// =========

bits(N) FPThree(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '1':Zeros(E-1);
frac = '1':Zeros(F-1);
return sign : exp : frac;
```
Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```c
// FPToFixed()
// ===========
// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
assert N IN {16,32,64};
assert M IN {16,32,64};
assert fbits >= 0;
assert rounding != FPRounding_ODD;

// Unpack using fpcr to determine if subnormals are flushed-to-zero
(type,sign,value) = FPUnpack(op, fpcr);

// If NaN, set cumulative flag or take exception
if type == FPType_SNaN || type == FPType_QNaN then
    FPProcessException(FPExc_InvalidOp, fpcr);

// Scale by fractional bits and produce integer rounded towards minus-infinity
value = value * 2.0^fbits;
int_result = RoundDown(value);
error = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment
case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_result > '1'));
    when FPRounding_POSINF
        round_up = (error != 0.0);
    when FPRounding_NEGINF
        round_up = FALSE;
    when FPRounding_ZERO
        round_up = (error != 0.0 && int_result < 0);
    when FPRounding_TIEAWAY
        round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
if round_up then int_result = int_result + 1;

// Generate saturated result and exceptions
(result, overflow) = SatQ(int_result, M, unsigned);
if overflow then
    FPProcessException(FPExc_InvalidOp, fpcr);
elsif error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);
return result;
```

Shared Pseudocode Functions
// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

bits(N) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64)

assert M == 64 && N == 32;

// Unpack using fpcr to determine if subnormals are flushed-to-zero
(type,sign,value) = FPUnpack(op, fpcr);

Z = '1';
// If NaN, set cumulative flag or take exception
if type == FPType_SNaN || type == FPType_QNaN then
    FPProcessException(FPExc_InvalidOp, fpcr);
    Z = '0';

int_result = RoundDown(value);
error = value - Real(int_result);
// Determine whether supplied rounding mode requires an increment
round_it_up = (error != 0.0 && int_result < 0);
if round_it_up then int_result = int_result + 1;
if int_result < 0 then
    result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
else
    result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

// Generate exceptions
if int_result < -(2^31) || int_result > (2^31)-1 then
    FPProcessException(FPExc_InvalidOp, fpcr);
    Z = '0';
elsif error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);
    Z = '0';
if sign == '1'&& value == 0.0 then
    Z = '0';
if type == FPType_Infinity then result = 0;
if Is64 then
    PSTATE.<N,Z,C,V> = '0':Z:'00';
else
    FPSCR<31:28> = '0':Z:'00';
return result<N-1:0>;

Library pseudocode for shared/functions/float/fptwo/FPTwo

// FPTwo()
// =======

bits(N) FPTwo(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '1':Zeros(E-1);
frac = Zeros(F);
return sign : exp : frac;
Library pseudocode for shared/functions/float/fptype/FPTypen

enumeration FPTypen {FPTypen_Nonzero, FPTypen_Zero, FPTypen_Infinity, FPTypen_QNaN, FPTypen_SNaN};

Library pseudocode for shared/functions/float/fpunpack/FPUnpack

// FPUnpack()
// =========
// 
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPTypen, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
   fpcr.AHP = '0';
   (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
   return (fp_type, sign, value);
// FPUnpackBase()
// ==============
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)
assert N IN {16,32,64};
if N == 16 then
    sign   = fpval<15>;
    exp16  = fpval<14:10>;
    frac16 = fpval<9:0>;
    if IsZero(exp16) then
        // Produce zero if value is zero or flush-to-zero is selected
        if IsZero(frac16) || fpcr.FZ16 == '1' then
            type = FPTYPE_Zero;  value = 0.0;
        else
            type = FPTYPE_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
    elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
        if IsZero(frac16) then
            type = FPTYPE_Infinity;  value = 2.0^1000000;
        else
            type = if frac16<9> == '1' then
                FPTYPE_QNaN
            else
                FPTYPE_SNaN;
            value = 0.0;
        else
            type = FPTYPE_Nonzero;
            value = 2.0^((UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10));
    endif
elsif N == 32 then
    sign   = fpval<31>;
    exp32  = fpval<30:23>;
    frac32 = fpval<22:0>;
    if IsZero(exp32) then
        // Produce zero if value is zero or flush-to-zero is selected.
        if IsZero(frac32) || fpcr.FZ == '1' then
            type = FPTYPE_Zero;  value = 0.0;
        else
            if !IsZero(frac32) then  // Denormalized input flushed to zero
                FPProcessException(FPExc_InputDenorm, fpcr);
            else
                type = FPTYPE_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
            endif
        else
            type = FPTYPE_Infinity;  value = 2.0^1000000;
        endif
        type = if frac32<22> == '1' then
            FPTYPE_QNaN
        else
            FPTYPE_SNaN;
        value = 0.0;
    else
        type = FPTYPE_Nonzero;
        value = 2.0^((UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23));
    endif
else // N == 64
    sign   = fpval<63>;
    exp64  = fpval<62:52>;
    frac64 = fpval<51:0>;
    if IsZero(exp64) then
        // Produce zero if value is zero or flush-to-zero is selected.
        if IsZero(frac64) || fpcr.FZ == '1' then
            type = FPTYPE_Zero;  value = 0.0;
        else
            if !IsZero(frac64) then  // Denormalized input flushed to zero
                FPProcessException(FPExc_InputDenorm, fpcr);
            else
                type = FPTYPE_Nonzero;
                value = 2.0^((UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52));
            endif
        endif
    else
        type = FPTYPE_Infinity;  value = 2.0^1000000;
    endif
end
type = FPTYPE_Nonzero; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
elsif IsOnes(exp64) then
  if IsZero(frac64) then
    type = FPTYPE_Infinity; value = 2.0^1000000;
  else
    type = if frac64<51> == '1' then FPTYPE_QNaN else FPTYPE_SNaN;
    value = 0.0;
  else
    type = FPTYPE_Nonzero;
    value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
  fi
fi
if sign == '1' then value = -value;
return (type, sign, value);

Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

// FPUnpackCV()
// -------------
// //
// // Used for FP <-> FP conversion instructions.
// // For half-precision data ignores FZ16 and observes AHP.
// (FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr)
// fpcr.FZ16 = '0';
// (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
// return (fp_type, sign, value);

Library pseudocode for shared/functions/float/fpzero/FPZero

// FPZero()
// -------
bits(N) FPZero(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp  = Zeros(E);
frac = Zeros(F);
return sign : exp : frac;

Library pseudocode for shared/functions/float/vfexpandimm/VFPExpandImm

// VFPExpandImm()
// ---------------
bits(N) VFPExpandImm(bits(8) imm8)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - E - 1;
sign = imm8<7>;
exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
frac = imm8<3:0>:Zeros(F-4);
return sign : exp : frac;
Library pseudocode for shared/functions/integer/AddWithCarry

```plaintext
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

<bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>;  // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

Library pseudocode for shared/functions/memory/AArch64.BranchAddr

```plaintext
// AArch64.BranchAddr()
// ================
// Return the virtual address with tag bits removed for storing to the program counter.

<bits(64) AArch64.BranchAddr(bits(64) vaddress)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>);
    else
        return ZeroExtend(vaddress<msbit:0>);
```

Library pseudocode for shared/functions/memory/AccType

```plaintext
enumeration AccType {
    AccType_NORMAL, AccType_VEC,  // Normal loads and stores
    AccType_STREAM, AccType_VECSTREAM,  // Streaming loads and stores
    AccType_ATOMIC, AccType_ATOMICRW,  // Atomic loads and stores
    AccType_ORDERED, AccType_ORDEREDRW,  // Load-Acquire and Store-Release
    AccType_ORDEREDDATOMIC,  // Load-Acquire and Store-Release with atomic
    AccType_ORDEREDDATOMICRW,
    AccType_LIMTEDORDERED,
    AccType_ATOMICRW,  // Load-LOAcquire and Store-LORelease
    AccType_UNPRIV,
    AccType_IFETCH,
    AccType_PTW,
    AccType_NONFAULT,
    AccType_CNOTFIRST,
    AccType_NV2REGISTER,  // MRS/MSR instruction used at EL1 and which is
                          // to a memory access that uses the EL2 translation
                          // Other operations
    AccType_DC,
    AccType_DC_UNPRIV,
    AccType_IC,
    AccType_DCZVA,
    AccType_AT};  // Address translation
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```plaintext
type AccessDescriptor is (  
    AccType acctype,
    MPAMinfo mpm,
    boolean page_table_walk,
    boolean secondstage,
    boolean s2fs1walk,
    integer level)
```

Shared Pseudocode Functions
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
assert HaveEL(el);
regime = S1TranslationRegime(el);
if ELUsingAArch32(regime) then
    // AArch32 translation regime.
    return 31;
else
    // AArch64 translation regime.
    case regime of
    when EL1
        tbi = (if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0);
        if HavePACExt() then
            tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        else
            tbi = TCR_EL1.TBI;
            if HavePACExt() then tbid = TCR_EL1.TBID;
    when EL2
        if HaveVirHostExt() && ELIsInHost(el) then
            tbi = (if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0);
            if HavePACExt() then
                tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if HavePACExt() then tbid = TCR_EL2.TBID;
        else
            tbi = TCR_EL2.TBI;
            if HavePACExt() then tbid = TCR_EL2.TBID;
    when EL3
        tbi = TCR_EL3.TBI;
        if HavePACExt() then tbid = TCR_EL3.TBID;
    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr ) then 55 else 63);

Library pseudocode for shared/functions/memory/AddressDescriptor

Library pseudocode for shared/functions/memory/AddressWithAllocationTag

Library pseudocode for shared/functions/memory/Allocation

Shared Pseudocode Functions
Library pseudocode for shared/functions/memory/AllocationTagFromAddress

```cpp
// AllocationTagFromAddress()
// =========================
// Generate a Tag from a 64-bit value containing a Logical Address Tag.
// If access to Allocation Tags is disabled, this function returns "00000000".

bits(4) AllocationTagFromAddress(bits(64) tagged_address)
    bits(4) logical_tag = tagged_address<59:56>;
    bits(4) tag = logical_tag + ('000':tagged_address<55>);
    return tag;
```

Library pseudocode for shared/functions/memory/BigEndian

```cpp
// BigEndian()
// ===========

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

Library pseudocode for shared/functions/memory/BigEndianReverse

```cpp
// BigEndianReverse()
// ==================

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) :
        BigEndianReverse(value<width-1:half>);
```

Library pseudocode for shared/functions/memory/Cacheability

```cpp
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

Library pseudocode for shared/functions/memory/CheckTag

```cpp
// CheckTag()
// =========
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean CheckTag(AddressDescriptor memaddrdesc, bits(4) ptag, boolean write)
    if memaddrdesc.memattrs.tagged then
        bits(64) paddress = ZeroExtend(memaddrdesc.paddress.address);
        return ptag == MemTag[paddress];
    else
        return TRUE;
```
Library pseudocode for shared/functions/memory/CreateAccessDescriptor

// CreateAccessDescriptor()
// ========================

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype IN {AccType_IFETCH, AccType_IC});
    accdesc.page_table_walk = FALSE;
    return accdesc;

Library pseudocode for shared/functions/memory/CreateAccessDescriptorPTW

// CreateAccessDescriptorPTW() // --------------------------

AccessDescriptor CreateAccessDescriptorPTW(AccType acctype, boolean secondstage, boolean s2fs1walk, integer level)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype IN {AccType_IFETCH, AccType_IC});
    accdesc.page_table_walk = TRUE;
    accdesc.secondstage = s2fs1walk;
    accdesc.secondstage = secondstage;
    accdesc.level = level;
    return accdesc;

Library pseudocode for shared/functions/memory/DataMemoryBarrier

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);

Library pseudocode for shared/functions/memory/DescriptorUpdate

type DescriptorUpdate is (  
    boolean AF, // AF needs to be set  
    AddressDescriptor descaddr // Descriptor to be updated  
)

Library pseudocode for shared/functions/memory/DeviceType

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
Library pseudocode for shared/functions/memory/EffectiveTBI

// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
assert HaveEL(el);
regime = S1TranslationStage(el);
assert(!ELUsingAArch32(regime));

(case regime of
  when EL1
    tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
    if HavePACExt() then
      tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
  when EL2
    if HaveVirtHostExt() && ELIsInHost(el) then
      tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
      if HavePACExt() then
        tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
    else
      tbi = TCR_EL2.TBI;
      if HavePACExt() then tbid = TCR_EL2.TBID;
  when EL3
    tbi = TCR_EL3.TBI;
    if HavePACExt() then tbid = TCR_EL3.TBID;
) return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');

Library pseudocode for shared/functions/memory/EffectiveTCMA

// EffectiveTCMA()
// ===============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
assert HaveEL(el);
regime = S1TranslationStage(el);
assert(!ELUsingAArch32(regime));

(case regime of
  when EL1
    tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
  when EL2
    if HaveVirtHostExt() && ELIsInHost(el) then
      tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
    else
      tcma = TCR_EL2.TCMA;
  when EL3
    tcma = TCR_EL3.TCMA;
)
return tcma;
library pseudocode for shared/functions/memory/Fault

```c
// Fault enumeration
```

library pseudocode for shared/functions/memory/FaultRecord

```c
type FaultRecord is (Fault type,  // Fault Status
                    AccType acctype,  // Type of access that faulted
                    FullAddress ipaddress,  // Intermediate physical address
                    boolean s2fs1walk,  // Is on a Stage 1 page table walk
                    boolean write,  // TRUE for a write, FALSE for a read
                    integer level,  // For translation, access flag and permission faults
                    bit extflag,  // IMPLEMENTATION DEFINED syndrome for external aborts
                    boolean secondstage,  // Is a Stage 2 abort
                    bits(4) domain,  // Domain number, AArch32 only
                    bits(2) errortype,  // [ARMv8.2 RAS] AArch32 AET or AArch64 SET
                    bits(4) debugmoe)  // Debug method of entry, from AArch32 only

library pseudocode for shared/functions/memory/FullAddress

```c
type FullAddress is (bits(52) address,  // '0' = Secure, '1' = Non-secure
                    bit NS)
```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.

Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);

// Definitions:

library pseudocode for shared/functions/memory/MBReqDomain

class enumeration MBReqDomain {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
MBReqDomain_OuterShareable, MBReqDomain_FullSystem};

library pseudocode for shared/functions/memory/MBReqTypes

class enumeration MBReqTypes {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};

library pseudocode for shared/functions/memory/MemAttrHints

class type MemAttrHints is (bits(2) attrs, // See MemAttr_*, Cacheability attributes
bits(2) hints, // See MemHint_*, Allocation hints
boolean transient
);
Library pseudocode for shared/functions/memory/MemTag

// MemTag[] - non-assignment (read) form
//=----------------------------------------------------------------------=
// Load an Allocation Tag from memory.

bits(4) MemTag[bits(64) address]
  AddressDescriptor memaddrdesc;
bits(4) value;
iswrite = FALSE;

  memaddrdesc = AArch64.TranslateAddress(address, AccType_NORMAL, iswrite, TRUE, TAG_GRANULE);
  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
  // Return the granule tag if tagging is enabled...
  if AllocationTagAccessIsEnabled() then
    return _MemTag[memaddrdesc];
  else
    // ...otherwise read tag as zero.
    return '0000';

// MemTag[] - assignment (write) form
//=----------------------------------------------------------------------=
// Store an Allocation Tag to memory.

MemTag[bits(64) address] = bits(4) value
  AddressDescriptor memaddrdesc;
iswrite = TRUE;

  // Stores of allocation tags must be aligned
  if address != Align(address, TAG_GRANULE) then
    boolean secondstage = FALSE;
    AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, iswrite, secondstage));
  wasaligned = TRUE;
  memaddrdesc = AArch64.TranslateAddress(address, AccType_NORMAL, iswrite, wasaligned, TAG_GRANULE);
  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
  // Memory array access
  if AllocationTagAccessIsEnabled() then
    _MemTag[memaddrdesc] = value;

Library pseudocode for shared/functions/memory/MemType

  enumeration MemType {MemType_Normal, MemType_Device};

Library pseudocode for shared/functions/memory/MemoryAttributes

type MemoryAttributes is (  
  MemType type,
  DeviceType device,       // For Device memory types
  MemAttrHints inner,      // Inner hints and attributes
  MemAttrHints outer,      // Outer hints and attributes
  boolean tagged,          // Tagged access
  boolean shareable,
  boolean outershareable
)
Library pseudocode for shared/functions/memory/Permissions

type Permissions is (  
    bits(3) ap,  // Access permission bits
    bit     xn,  // Execute-never bit
    bit     xxn, // [ARMv8.2] Extended execute-never bit for stage 2
    bit     pxn // Privileged execute-never bit
)

Library pseudocode for shared/functions/memory/PrefetchHint

enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};

Library pseudocode for shared/functions/memory/SpeculativeSynchronizationBarrierToPA

SpeculativeSynchronizationBarrierToPA();

Library pseudocode for shared/functions/memory/SpeculativeSynchronizationBarrierToVA

SpeculativeSynchronizationBarrierToVA();

Library pseudocode for shared/functions/memory/TLBRecord

type TLBRecord is (  
    Permissions perms,  // '0' = Global, '1' = not Global
    bit nG,  // AArch32 only
    bits(4) domain,  // Contiguous bit from page table
    bit GP,  // Guarded Page
    boolean contiguous, // Contiguous bit from page table
    integer level,  // AArch32 Short-descriptor format: Indicates Section/Page
    integer blocksize,  // Describes size of memory translated in KBytes
    DescriptorUpdate descupdate,  // [ARMv8.1] Context for h/w update of table descriptor
    bit CnP,  // [ARMv8.2] TLB entry can be shared between different PEs
    AddressDescriptor addrdesc
)

Library pseudocode for shared/functions/memory/TransformTag

// TransformTag()
// ---------------
// Apply tag transformation rules.

bits(4) TransformTag(bits(64) vaddr)
    bits(4) vtag = vaddr<59:56>;
    bits(4) tagdelta = ZeroExtend(vaddr<55>);
    bits(4) ptag = vtag + tagdelta;
    return ptag;

Library pseudocode for shared/functions/memory/_Mem

// These two _Mem[] accessors are the hardware operations which perform single-copy atomic,  
// aligned, little-endian memory accesses of size bytes from/to the underlying physical  
// memory array of bytes.  
//  
// The functions address the array using desc.paddress which supplies:  
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.  
//  
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,  
// etc and other parameters required to access the physical memory or for setting syndrome  
// register in the event of an external abort.  
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc] = bits(8*size) value;
// boolean AccessIsTagChecked()
// ===============
// TRUE if a given access is tag-checked, FALSE otherwise.
boolean AccessIsTagChecked(bits(64) vaddr, AccType acctype)
    if PSTATE.M<4> == '1' then return FALSE;
    if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
        return FALSE;
    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') then
        return FALSE;
    if AllocationTagAccessIsEnabled() then
        return FALSE;
    if acctype IN {AccType_IFETCH, AccType_PTW} then
        return FALSE;
    if acctype == AccType_NV2REGISTER then
        return FALSE;
    if PSTATE.TCO=='1' then
        return FALSE;
    if IsNonTagCheckedInstruction() then
        return FALSE;
    return TRUE;

Library pseudocode for shared/functions/mpam/DefaultMPAMinfo

// DefaultMPAMinfo
// ===============
// Returns default MPAM info. If secure is TRUE return default Secure MPAMinfo, otherwise return default Non-secure MPAMinfo.
MPAMinfo DefaultMPAMinfo(boolean secure)
    MPAMinfo DefaultInfo;
    DefaultInfo.mpam_NS = if secure then '0' else '1';
    DefaultInfo.partid  = DefaultPARTID;
    DefaultInfo.pmg     = DefaultPMG;
    return DefaultInfo;

Library pseudocode for shared/functions/mpam/DefaultPARTID

constant PARTIDtype DefaultPARTID = 0<15:0>;

Library pseudocode for shared/functions/mpam/DefaultPMG

constant PMGtype DefaultPMG = 0<7:0>;
// GenMPAMcurEL
// ============
// Returns MPAMinfo for the current EL and security state.
// InD is TRUE instruction access and FALSE otherwise.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(boolean InD)
bits(2) mpamel;
boolean validEL;
boolean secure = IsSecure();
if HaveMPAMExt() && MPAMisEnabled() then
    if UsingAArch32() then
        (validEL, mpamel) = ELFromM32(PSTATE.M);
    else
        validEL = TRUE;
        mpamel = PSTATE.EL;
    if validEL then
        return genMPAM(UInt(mpamel), InD, secure);
return DefaultMPAMinfo(secure);
Library pseudocode for shared/functions/mpam/MAP_vPARTID

// MAP_vPARTID
// ===========
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
// should not ever be called if EL2 is not implemented
// or is implemented but not enabled in the current
// security state.
PARTIDtype ret;
boolean err;
integer virt    = UInt( vpartid );
integer vmprmax = UInt( MPAMIDR_EL1.VPMR_MAX );

// vpartid_max is largest vpartid supported
integer vpartid_max = 4 * vmprmax + 3;

// One of many ways to reduce vpartid to value less than vpartid_max.
if virt > vpartid_max then
  virt = virt MOD (vpartid_max+1);

// Check for valid mapping entry.
if MPAMVPMV_EL2<virt> == '1' then
  // vpartid has a valid mapping so access the map.
  ret = mapvpmw(virt);
  err = FALSE;
elseif MPAMVPMV_EL2<0> == '1' then
  // Yes, so use default mapping for vpartid == 0.
  ret = MPAMVPM0_EL2<0 +: 16>;
  err = FALSE;
else
  ret = DefaultPARTID;
  err = TRUE;

// Check that the physical PARTID is in-range.
// This physical PARTID came from a virtual mapping entry.
integer partid_max = UInt( MPAMIDR_EL1.PARTID_MAX );
if UInt(ret) > partid_max then
  // Out of range, so return default physical PARTID
  ret = DefaultPARTID;
  err = TRUE;
return (ret, err);

Library pseudocode for shared/functions/mpam/MPAMisEnabled

// MPAMisEnabled
// =============
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
  el = HighestEL();
  case el of
    when EL3  return MPAM3_EL3.MPAMEN == '1';
    when EL2  return MPAM2_EL2.MPAMEN == '1';
    when EL1  return MPAM1_EL1.MPAMEN == '1';
Library pseudocode for shared/functions/mpam/MPAMisVirtual

```c
// MPAMisVirtual
// =============
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(integer el)
    return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
            (el == 0 && MPAMHCR_EL2.EL0_VPMEN == '1') ||
            (el == 1 && MPAMHCR_EL2.EL1_VPMEN == '1'));
```

Library pseudocode for shared/functions/mpam/genMPAM

```c
// genMPAM
// ========
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// Produces a Secure PARTID if Secure is TRUE and a Non-secure PARTID otherwise.

MPAMinfo genMPAM(integer el, boolean InD, boolean secure)
    MPAMinfo returnInfo;
    PARTIDtype partidel;
    boolean perr;
    boolean gstplk = (el == 0 && EL2Enabled() &&
                      MPAMHCR_EL2.GSTAPP_PLK == '1' && HCR_EL2.TGE == '0');
    integer eff_el = if gstplk then 1 else el;
    (partidel, perr) = genPARTID(eff_el, InD);
    PMGtype groupel = genPMG(eff_el, InD, perr);
    returnInfo.mpam_ns = if secure then '0' else '1';
    returnInfo.partid = partidel;
    returnInfo.pmg = groupel;
    return returnInfo;
```

Library pseudocode for shared/functions/mpam/genMPAMEl

```c
// genMPAMEl
// =========
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.

MPAMinfo genMPAMEl(bits(2) el, boolean InD)
    boolean secure = IsSecure();
    if HaveMPAMExt() && MPAMisEnabled() then
        return genMPAM(UInt(el), InD, secure);
    return DefaultMPAMinfo(secure);
```

Library pseudocode for shared/functions/mpam/genPARTID

```c
// genPARTID
// =========
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.

(PARTIDtype, boolean) genPARTID(integer el, boolean InD)
    PARTIDtype partidel = getMPAM_PARTID(el, InD);
    integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(partid_max) > partid_max then
        return (DefaultPARTID, TRUE);
    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```
Library pseudocode for shared/functions/mpam/genPMG

// genPMG
// ========
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(integer el, boolean InD, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
        PMGtype groupel = getMPAM_PMG(el, InD);
        if UInt(groupel) <= pmg_max then
            return groupel;
        return DefaultPMG;

Library pseudocode for shared/functions/mpam/getMPAM_PARTID

// getMPAM_PARTID
// ==============
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(integer MPAMn, boolean InD)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();
    if InD then
        case MPAMn of
            when 3 partid = MPAM3_EL3.PARTID_I;
            when 2 partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
            when 1 partid = MPAM1_EL1.PARTID_I;
            when 0 partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
        else
            case MPAMn of
                when 3 partid = MPAM3_EL3.PARTID_D;
                when 2 partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros();
                when 1 partid = MPAM1_EL1.PARTID_D;
                when 0 partid = MPAM0_EL1.PARTID_D;
                otherwise partid = PARTIDtype UNKNOWN;
            return partid;
Library pseudocode for shared/functions/mpam/getMPAM_PMG

// getMPAM_PMG
// ===========
// Returns a PMG from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(integer MPAMn, boolean InD)
PMGtype pmg;
boolean el2avail = EL2Enabled();
if InD then
  case MPAMn of
    when 3 pmg = MPAM3_EL3.PMG_I;
    when 2 pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros();
    when 1 pmg = MPAM1_EL1.PMG_I;
    when 0 pmg = MPAM0_EL1.PMG_I;
    otherwise pmg = PMGtype UNKNOWN;
else
  case MPAMn of
    when 3 pmg = MPAM3_EL3.PMG_D;
    when 2 pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros();
    when 1 pmg = MPAM1_EL1.PMG_D;
    when 0 pmg = MPAM0_EL1.PMG_D;
    otherwise pmg = PMGtype UNKNOWN;
return pmg;

Library pseudocode for shared/functions/mpam/mapvpmw

// mapvpmw
// =======
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
bits(64) vpmw;
integer wd = vpartid DIV 4;
case wd of
  when 0 vpmw = MPAMVPM0_EL2;
  when 1 vpmw = MPAMVPM1_EL2;
  when 2 vpmw = MPAMVPM2_EL2;
  when 3 vpmw = MPAMVPM3_EL2;
  when 4 vpmw = MPAMVPM4_EL2;
  when 5 vpmw = MPAMVPM5_EL2;
  when 6 vpmw = MPAMVPM6_EL2;
  when 7 vpmw = MPAMVPM7_EL2;
  otherwise vpmw = Zeros(64);
// vpme_lsb selects LSB of field within register
integer vpme_lsb = (vpartid REM 4) * 16;
return vpmw<vpme_lsb +: 16>;
// BranchTo()
// =========
// Set program counter to a new address, with a branch type
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type)

    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = AArch64.BranchAddr(target<63:0>);
    return;

// BranchToAddr()
// =============
// Set program counter to a new address, with a branch type
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)

    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;

Library pseudocode for shared/functions/registers/BranchType

<table>
<thead>
<tr>
<th>BranchType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BranchType_DIRCALL</td>
<td>Direct Branch with link</td>
</tr>
<tr>
<td>BranchType_INDCALL</td>
<td>Indirect Branch with link</td>
</tr>
<tr>
<td>BranchType_ERET</td>
<td>Exception return (indirect)</td>
</tr>
<tr>
<td>BranchType_DBGEXIT</td>
<td>Exit from Debug state</td>
</tr>
<tr>
<td>BranchType_RET</td>
<td>Indirect branch with function return hint</td>
</tr>
<tr>
<td>BranchType_DIR</td>
<td>Direct branch</td>
</tr>
<tr>
<td>BranchType_INDIR</td>
<td>Indirect branch</td>
</tr>
<tr>
<td>BranchType_EXCEPTION</td>
<td>Exception entry</td>
</tr>
<tr>
<td>BranchType_RESET</td>
<td>Reset</td>
</tr>
<tr>
<td>BranchType_UNKNOWN</td>
<td>Other</td>
</tr>
</tbody>
</table>

// Hint_Branch()
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.

// NextInstrAddr()
// Return address of the sequentially next instruction.

// ResetExternalDebugRegisters()
// Reset the External Debug registers in the Core power domain.
Library pseudocode for shared/functions/registers/ThisInstrAddr

```c
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr()
assert N == 64 || (N == 32 && UsingAArch32());
return _PC<N-1:0>;
```

Library pseudocode for shared/functions/registers/_PC

```c
bits(64) _PC;
```

Library pseudocode for shared/functions/registers/_R

```c
array bits(64) _R[0..30];
```

Library pseudocode for shared/functions/registers/_V

```c
array bits(128) _V[0..31];
```
// SPSR[] - non-assignment form
// -----------------------------------

bits(32) SPSR[]
bits(32) result;
if UsingAArch32() then
    case PSTATE.M of
    when M32_FIQ    result = SPSR_fiq;
    when M32_IRQ     result = SPSR_irq;
    when M32_Svc     result = SPSR_svc;
    when M32_Monitor result = SPSR_mon;
    when M32_Abort   result = SPSR_abt;
    when M32_Hyp     result = SPSR_hyp;
    when M32.Undef   result = SPSR_und;
    otherwise Unreachable();
else
    case PSTATE.EL of
    when EL1         result = SPSR_EL1;
    when EL2         result = SPSR_EL2;
    when EL3         result = SPSR_EL3;
    otherwise Unreachable();
return result;
// SPSR[] - assignment form
// -----------------------------------

SPSR[] = bits(32) value
if UsingAArch32() then
    case PSTATE.M of
    when M32_FIQ     SPSR_fiq = value;
    when M32_IRQ      SPSR_irq = value;
    when M32_Svc      SPSR_svc = value;
    when M32_Monitor  SPSR_mon = value;
    when M32_Abort    SPSR_abt = value;
    when M32_Hyp      SPSR_hyp = value;
    when M32.Undef    SPSR_und = value;
    otherwise Unreachable();
else
    case PSTATE.EL of
    when EL1         SPSR_EL1 = value;
    when EL2         SPSR_EL2 = value;
    when EL3         SPSR_EL3 = value;
    otherwise Unreachable();
return;
// AllocationTagAccessIsEnabled()
// ---------------------------------------------
// Check whether access to Allocation Tags is enabled.

boolean AllocationTagAccessIsEnabled()
    if SCR_EL3.ATA == '0' && PSTATE.EL IN {EL0, EL1, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && HCR_EL2.<E2H,TGE> != '11' && PSTATE.EL IN {EL0, EL1} then
        return FALSE;
    elsif SCTLR_EL3.ATA == '0' && PSTATE.EL == EL3 then
        return FALSE;
    elsif SCTLR_EL2.ATA == '0' && PSTATE.EL == EL2 then
        return FALSE;
    elsif SCTLR_EL1.ATA == '0' && PSTATE.EL == EL1 then
        return FALSE;
    elsif SCTLR_EL2.ATA0 == '0' && HCR_EL2.<E2H,TGE> == '11' && PSTATE.EL == EL0 then
        return FALSE;
    elsif SCTLR_EL1.ATA0 == '0' && HCR_EL2.<E2H,TGE> != '11' && PSTATE.EL == EL0 then
        return FALSE;
    else
        return TRUE;

Library pseudocode for shared/functions/system/ArchVersion

enumeration ArchVersion {
    ARMv8p0 , ARMv8p1 , ARMv8p2 , ARMv8p3 , ARMv8p4 , ARMv8p5
};

Library pseudocode for shared/functions/system/BranchTargetCheck

// BranchTargetCheck()
// -------------------
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()
    assert HaveBTIExt() && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        bits(64) pc = ThisInstrAddr();
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBROrBLORetInstr();
    boolean bti_instr    = AArch64.ExecutingBTIInstr();

    // PSTATE.BTYPE defaults to 00 for instructions that don't explicitly set BTYPE.
    if (!branch_instr || bti_instr) then
        BTypeNext = '00';
// ChooseNonExcludedTag()
// ===============
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) ChooseNonExcludedTag(bits(4) tag, bits(4) offset, bits(16) exclude)
    if exclude == Ones(16) then
        return tag;
    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';
        return tag;

Library pseudocode for shared/functions/system/ClearEventRegister

// ClearEventRegister()
// ===============
// Clear the Event Register of this PE

ClearEventRegister()
    EventRegister = '0';
    return;

Library pseudocode for shared/functions/system/ClearPendingPhysicalSError

// Clear a pending physical SError interrupt
ClearPendingPhysicalSError();

Library pseudocode for shared/functions/system/ClearPendingVirtualSError

// Clear a pending virtual SError interrupt
ClearPendingVirtualSError();

Library pseudocode for shared/functions/system/ConditionHolds

// ConditionHolds()
// ================
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1'); // EQ or NE
        when '001' result = (PSTATE.C == '1'); // CS or CC
        when '010' result = (PSTATE.N == '1'); // MI or PL
        when '011' result = (PSTATE.V == '1'); // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V); // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
        when '111' result = TRUE; // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;
    return result;
Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

ConsumptionOfSpeculativeDataBarrier();

Library pseudocode for shared/functions/system/CurrentInstrSet

// CurrentInstrSet()
// -------------

InstrSet CurrentInstrSet()

    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
        else
            result = InstrSet_A64;
        return result;
    else
        result = InstrSet_A64;
        return result;

Library pseudocode for shared/functions/system/CurrentPL

// CurrentPL()
// -------------

PrivilegeLevel CurrentPL()

    return PLoFPL(PSTATE.EL);

Library pseudocode for shared/functions/system/EL0

    constant bits(2) EL3 = '11';
    constant bits(2) EL2 = '10';
    constant bits(2) EL1 = '01';
    constant bits(2) EL0 = '00';

Library pseudocode for shared/functions/system/EL2Enabled

// EL2Enabled()
// -------------

// Returns TRUE if EL2 is present and executing in either non-Secure state when Secure EL2 is not implemented,
// or in Secure state when Secure EL2 is implemented, FALSE otherwise

boolean EL2Enabled()

    return IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure());
**Library pseudocode for shared/functions/system/ELFromM32**

// ELFromM32()
// ===========

(boolean, bits(2)) ELFromM32(bits(5) mode)

// Convert an AArch32 mode encoding to an exception level.
// Returns (valid, EL):
// 'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
// and the current value of SCR.NS/SCR_EL3.NS.
// 'EL' is the exception level decoded from 'mode'.

bits(2) el;
boolean valid = !BadMode(mode);  // Check for modes that are not valid for this implementation

case mode of
  when M32_Monitor
    el = EL3;
  when M32_Hyp
    el = EL2;
  when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
    valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
  when M32_User
    el = EL0;
  otherwise
    valid = FALSE;           // Passed an illegal mode value

if !valid then el = bits(2) UNKNOWN;
return (valid, el);

**Library pseudocode for shared/functions/system/ELFromSPSR**

// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an exception level.
// Returns (valid, EL):
// 'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
// 'EL' is the exception level decoded from 'spsr'.

(boolean, bits(2)) ELFromSPSR(bits(32) spsr)

if spsr<4> == '0' then  // AArch64 state
  el = spsr<3:2>;
else if HighestELUsingAArch32() then  // No AArch64 support
  valid = FALSE;
elseif !HaveEL(el) then  // Exception level not implemented
  valid = FALSE;
elseif spsr<1> == '1' then  // M[1] must be 0
  valid = FALSE;
elseif el == EL0 && spsr<0> == '1' then  // for EL0, M[0] must be 0
  valid = FALSE;
elseif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
  valid = FALSE;  // Unless Secure EL2 is enabled, EL2 only valid in Non-
else
  valid = TRUE;
elseif !HaveAnyAArch32() then  // AArch32 not supported
  valid = FALSE;
else  // AArch32 state
  (valid, el) = ELFromM32(spsr<4:0>);
if !valid then el = bits(2) UNKNOWN;
return (valid, el);
// ELIsInHost()
// ============
boolean ELIsInHost(bits(2) el)
    return ((IsSecureEL2Enabled() || !IsSecureBelowEL3()) && HaveVirtHostExt() && !ELUsingAArch32(EL2)
            HCR_EL2.E2H == '1' && (el == EL2 || (el == EL0 && HCR_EL2.TGE == '1')));

// ELStateUsingAArch32()
// =====================
boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;

// ELStateUsingAArch32K()
// ======================
(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    return
        known = TRUE;
        if !HaveAArch32EL(el) then
            aarch32 = FALSE;                           // Exception level is using AArch64
        elsif HighestELUsingAArch32() then
            aarch32 = TRUE;                            // All levels are using AArch32
        else
            aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0';
            aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) &&
                (HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) && HCR_EL2.RW == '0'
                && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt()));
            if el == EL0 && !aarch32_at_el1 then       // Only know if EL0 using AArch32 from PSTATE
                if PSTATE.EL == EL0 then
                    aarch32 = PSTATE.nRW == '1';       // EL0 controlled by PSTATE
                else
                    known = FALSE;                     // EL0 state is UNKNOWN
            else
                aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});
            end if
        end if
    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);

// ELUsingAArch32()
// ================
boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
Library pseudocode for shared/functions/system/ELUsingAArch32K

// ELUsingAArch32K()
// ===============

(boolean,boolean) ELUsingAArch32K(bits(2) el)
return ELStateUsingAArch32K(el, IsSecureBelowEL3());

Library pseudocode for shared/functions/system/EndOfInstruction

// Terminate processing of the current instruction.
EndOfInstruction();

Library pseudocode for shared/functions/system/EnterLowPowerState

// PE enters a low-power state
EnterLowPowerState();

Library pseudocode for shared/functions/system/EventRegister

bits(1) EventRegister;

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

// GetPSRFromPSTATE()
// ================

// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
bits(32) spsr = Zeros();
spsr<31:28> = PSTATE.<N,Z,C,V>;
if HaveDITExt() then spsr<24> = PSTATE.DIT;
if HavePANExt() then spsr<22> = PSTATE.PAN;
spsr<21> = PSTATE.SS;
spsr<20> = PSTATE.IL;
if PSTATE.nRW == '1' then // AArch32 state
spsr<27> = PSTATE.Q;
spsr<26:25> = PSTATE.IT<1:0>;
if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
spsr<19:16> = PSTATE.GE;
spsr<15:10> = PSTATE.IT<7:2>;
spsr<9:8> = PSTATE.E;
spsr<8:6> = PSTATE.<A,I,F>;
if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
spsr<5> = PSTATE.T;
assert PSTATE.M<4> == PSTATE.nRW; // No PSTATE.D in AArch32 state
spsr<4:0> = PSTATE.M;
else // AArch64 state
if HaveUAOExt() then spsr<23> = PSTATE.UAO;
if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
if HaveMTEExt() then spsr<25> = PSTATE.TCO;
if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
spsr<9:6> = PSTATE.<D,A,I,F>;
spsr<4> = PSTATE.nRW;
spsr<3:2> = PSTATE.EL;
spsr<0> = PSTATE.SP;
return spsr;

Library pseudocode for shared/functions/system/HasArchVersion

// HasArchVersion()
// ================

// Return TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
// HaveAArch32EL()
// ===============

boolean HaveAArch32EL(bits(2) el)
// Return TRUE if Exception level 'el' supports AArch32 in this implementation
if !HaveEL(el) then
    return FALSE;                    // The Exception level is not implemented
elsif !HaveAnyAArch32() then
    return FALSE;                    // No Exception level can use AArch32
elsif HighestELUsingAArch32() then
    return TRUE;                     // All Exception levels are using AArch32
elsif el == HighestEL() then
    return FALSE;                    // The highest Exception level is using AArch64
elsif el == EL0 then
    return TRUE;                     // EL0 must support using AArch32 if any AArch32
return boolean IMPLEMENTATION_DEFINED;

// HaveAnyAArch32()
// ================

// Return TRUE if AArch32 state is supported at any Exception level
boolean HaveAnyAArch32()
return boolean IMPLEMENTATION_DEFINED;

// HaveAnyAArch64()
// ===============

// Return TRUE if AArch64 state is supported at any Exception level
boolean HaveAnyAArch64()
return !HighestELUsingAArch32();

// HaveEL()
// =========

// Return TRUE if Exception level 'el' is supported
boolean HaveEL(bits(2) el)
if el IN {EL1, EL0} then
    return TRUE;                             // EL1 and EL0 must exist
return boolean IMPLEMENTATION_DEFINED;

// HaveFP16Ext()
// =============

// Return TRUE if FP16 extension is supported
boolean HaveFP16Ext()
return boolean IMPLEMENTATION_DEFINED;
Library pseudocode for shared/functions/system/HighestEL

```plaintext
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

Library pseudocode for shared/functions/system/HighestELUsingAArch32

```plaintext
// HighestELUsingAArch32()
// =--------------------------------
// Return TRUE if configured to boot into AArch32 operation

boolean HighestELUsingAArch32()
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED;       // e.g. CFG32SIGNAL == HIGH
```

Library pseudocode for shared/functions/system/Hint_Yield

```plaintext
Hint_Yield();
```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(32) spsr)
{
    // Check for illegal return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state, when SecureEL2 is not enabled.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    // Execution state used in the Exception level being returned to, as determined by
    // the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    // * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    // SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    // * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !UsingAArch32K(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;
}

Library pseudocode for shared/functions/system/InstrSet

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

InstructionSynchronizationBarrier();

Library pseudocode for shared/functions/system/InterruptPending

// InterruptPending()
// -------------------
// Return TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise

boolean InterruptPending()
{
    return IsPhysicalSErrorPending() || IsVirtualSErrorPending();
}

Library pseudocode for shared/functions/system/IsEventRegisterSet

// IsEventRegisterSet()
// ---------------------
// Return TRUE if the Event Register of this PE is set, and FALSE otherwise

boolean IsEventRegisterSet()
{
    return EventRegister == '1';
}
Library pseudocode for shared/functions/system/IsHighestEL

```plaintext
// IsHighestEL()
// =============
// Returns TRUE if given exception level is the highest exception level implemented
boolean IsHighestEL(bits(2) el)
return HighestEL() == el;
```

Library pseudocode for shared/functions/system/IsInHost

```plaintext
// IsInHost()
// =---------
boolean IsInHost()
return ELIsInHost(PSTATE.EL);
```

Library pseudocode for shared/functions/system/IsPhysicalSErrorPending

```plaintext
// Return TRUE if a physical SError interrupt is pending
boolean IsPhysicalSErrorPending();
```

Library pseudocode for shared/functions/system/IsSecure

```plaintext
// IsSecure()
// =---------
boolean IsSecure()
// Return TRUE if current Exception level is in Secure state.
// or would be following an exception return to that level.
//
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
  return TRUE;
elsif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
  return TRUE;
return IsSecureBelowEL3();
```

Library pseudocode for shared/functions/system/IsSecureBelowEL3

```plaintext
// IsSecureBelowEL3()
// =----------------
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
if HaveEL(EL3) then
  return SCR_GEN[].NS == '0';
elsif HaveEL(EL2) && (!HaveSecureEL2Ext() || HighestELUsingAArch32()) then
  // If Secure EL2 is not an architecture option then we must be Non-secure.
  return FALSE;
else
  // TRUE if processor is Secure or FALSE if Non-secure.
  return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```
// IsSecureEL2Enabled()
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise

boolean IsSecureEL2Enabled() {
    return (HaveSecureEL2Ext() && HaveEL(EL2) && !ELUsingAArch32(EL2) && ((HaveEL(EL2) && !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1') || (!HaveEL(EL3) && IsSecure())));
}

// Return TRUE if a virtual SError interrupt is pending
boolean IsVirtualSErrorPending();

// PLOfEL()
// -------

PrivilegeLevel PLOfEL(bits(2) el) {
    case el of
        when EL3 return if HighestELUsingAArch32() then PL1 else PL3;
        when EL2 return PL2;
        when EL1 return PL1;
        when EL0 return PL0;
}

ProcState PSTATE;

enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
Library pseudocode for shared/functions/system/ProcState

type ProcState is (  
  bits (1) N,     // Negative condition flag  
  bits (1) Z,     // Zero condition flag  
  bits (1) C,     // Carry condition flag  
  bits (1) V,     // oVerflow condition flag  
  bits (1) D,     // Debug mask bit [AArch64 only]  
  bits (1) A,     // SError interrupt mask bit  
  bits (1) I,     // IRQ mask bit  
  bits (1) F,     // FIQ mask bit  
  bits (1) PAN,   // Privileged Access Never Bit [v8.1]  
  bits (1) UAO,   // User Access Override [v8.2]  
  bits (1) BTYPE, // Branch Type [v8.5]  
  bits (1) SS,    // Software step bit  
  bits (1) IL,    // Illegal Execution state bit  
  bits (2) EL,    // Exception Level  
  bits (1) nRW,   // not Register Width: 0=64, 1=32  
  bits (1) SP,    // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]  
  bits (1) Q,     // Cumulative saturation flag [AArch32 only]  
  bits (4) GE,    // Greater than or Equal flags [AArch32 only]  
  bits (1) SSBS,  // Speculative Store Bypass Safe  
  bits (8) IT,    // If-then bits, RES0 in CPSR [AArch32 only]  
  bits (1) J,     // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]  
  bits (1) T,     // T32 bit, RES0 in CPSR [AArch32 only]  
  bits (1) E,     // Endianness bit [AArch32 only]  
  bits (1) M      // Mode field [AArch32 only]  
)

Library pseudocode for shared/functions/system/RandomTag

  // RandomTag()  
  // ===========  
  // Generate a random Allocation Tag.

  bits(4) RandomTag()  
  bits(4) tag;  
  for i = 0 to 3  
    tag<i> = NextRandomTagBit();  
  return tag;

Library pseudocode for shared/functions/system/RandomTagBit

  // RandomTagBit()  
  // ==============  
  // Generate a random bit suitable for generating a random Allocation Tag.

  bit NextRandomTagBit()  
  bits(16) lfsr = RGSR_EL1.SEED;  
  bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;  
  RGSR_EL1.SEED = top:lfsr<15:1>;  
  return top;
// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(32) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && !IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;

Library pseudocode for shared/functions/system/SCRType

type SCRType;

Library pseudocode for shared/functions/system/SCR_GEN

// SCR_GEN[]
// =========

SCRType SCR_GEN[]

    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(32) r;
    if HighestELUsingAArch32() then
        r = SCR;
    else
        r = SCR_EL3;
    return r;

Library pseudocode for shared/functions/system/SendEvent

// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed
SendEvent();

Library pseudocode for shared/functions/system/SendEventLocal

// SendEventLocal()
// ================

SendEventLocal()
    EventRegister = '1';
    return;
Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```c
// SetPSTATEFromPSR()
// ================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then  // AArch32 state
            AArch32.WriteMode(spsr<4:0>);  // Sets PSTATE.EL correctly
        else  // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL = spsr<3:2>;
            PSTATE.SP = spsr<0>;
        // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
        // the T bit is set to zero or copied from SPSR.
        if PSTATE.IL == '1' & PSTATE.nRW == '1' then
            if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';
        // State that is reinstated regardless of illegal exception return
        PSTATE.<N,Z,C,V> = spsr<31:28>;
        if HaveDITExt() then PSTATE.DIT = spsr<24>;
        if PSTATE.nRW == '1' then  // AArch32 state
            PSTATE.Q = spsr<27>;
            ShouldAdvanceIT = FALSE;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
            PSTATE.GE = spsr<19:16>;
            PSTATE.E = spsr<9>;
            PSTATE.<A,I,F> = spsr<8:6>;  // No PSTATE.D in AArch32 state
            PSTATE.T = spsr<5>;
                // PSTATE.J is RES0
        else  // AArch64 state
            if HaveUAOExt() then PSTATE.UAO = spsr<23>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
            PSTATE.<D,A,I,F> = spsr<9:6>;
            if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
        if HaveFANExt() then PSTATE.PAN = spsr<22>;
        if HaveMTEExt() then
            if PSTATE.nRW != '1' then
                PSTATE.TCO = spsr<25>;
        return;
```

Library pseudocode for shared/functions/system/ShouldAdvanceIT

```c
boolean ShouldAdvanceIT;
```

Library pseudocode for shared/functions/system/SpeculationBarrier

```c
SpeculationBarrier();
```

Library pseudocode for shared/functions/system/SynchronizeContext

```c
SynchronizeContext();
```

Library pseudocode for shared/functions/system/SynchronizeErrors

```c
// Implements the error synchronization event.
SynchronizeErrors();
```
// Take any pending unmasked physical SError interrupt
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);

// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.
TakeUnmaskedSErrorInterrupts();

bits(32) ThisInstr();

integer ThisInstrLength();

Unreachable()
    assert FALSE;

// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.
boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAnyAArch32() then assert !aarch32;
    if HighestELUsingAArch32() then assert aarch32;
    return aarch32;

// WaitForEvent()
// ==============
// PE suspends its operation and enters a low-power state
// if the Event Register is clear when the WFE is executed
WaitForEvent()
    if EventRegister == '0' then
        EnterLowPowerState();
    return;

// WaitForInterrupt()
// ===============
// PE suspends its operation to enter a low-power state
// until a WFI wake-up event occurs or the PE is reset
WaitForInterrupt()
    EnterLowPowerState();
    return;
Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictable
// ConstraintUnpredictable()
// ----------------------------------------
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.

Constraint ConstraintUnpredictable(Unpredictable which)
    case which of
        when Unpredictable WBOVERLAPLD
            return Constraint WBSUPPRESS; // return loaded value
        when Unpredictable WBOVERLAPST
            return Constraint NONE; // store pre-writeback value
        when Unpredictable LDPOVERLAP
            return Constraint UNDEF; // instruction is UNDEFINED
        when Unpredictable BASEOVERLAP
            return Constraint NONE; // use original address
        when Unpredictable DATAOVERLAP
            return Constraint NONE; // store original value
        when Unpredictable DEVPAGE2
            return Constraint FAULT; // store value
        when Unpredictable INSTRDEVICE
            return Constraint NONE; // Do not take a fault
        when Unpredictable RESCPACR
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESMAIR
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESTEXCB
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESDACR
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESPRRR
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESVTCRS
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESTnSZ
            return Constraint FORCE; // Map to the limit value
        when Unpredictable OORTnSZ
            return Constraint FORCE; // Map to the limit value
        when Unpredictable LARGEIPA
            return Constraint FORCE; // Restrict the inputsize to the PAMax value
        when Unpredictable ESRCONDPASS
            return Constraint FALSE; // Report as "AL"
        when Unpredictable ILZEROIT
            return Constraint FALSE; // Do not zero PSTATE.IT
        when Unpredictable ILZEROT
            return Constraint FALSE; // Do not zero PSTATE.T
        when Unpredictable BPVECTORCATCHPRI
            return Constraint TRUE; // Debug Vector Catch: match on 2nd halfword
        when Unpredictable VCMATCHHALF
            return Constraint FALSE; // No match
        when Unpredictable VCMATCHDAPA
            return Constraint FALSE; // No match on Data Abort or Prefetch abort
        when Unpredictable WPMASKANDDBAS
            return Constraint FALSE; // Watchpoint disabled
        when Unpredictable WPBASCONTIGUOUS
            return Constraint FALSE; // Watchpoint disabled
        when Unpredictable RESWPMASK
            return Constraint DISABLED; // Watchpoint disabled
        when Unpredictable WMASKEDBITS
            return Constraint FALSE; // Watchpoint disabled
        when Unpredictable RESBWPCCTRL
            return Constraint DISABLED; // Breakpoint/watchpoint disabled
        when Unpredictable BPNOTIMPL
            return Constraint DISABLED; // Breakpoint disabled

Shared Pseudocode Functions
```
when Unpredictable_RESBPTYPE
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPMATCHHALF
    return Constraint_FALSE;    // No match
when Unpredictable_BPNOTCTXCMP
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPMISMATCHHALF
    return Constraint_FALSE;    // No match
when Unpredictable_RESTARTALIGNPC
    return Constraint_FALSE;    // Do not force alignment
when Unpredictable_RESTARTZEROUPPERPC
    return Constraint_TRUE;     // Force zero extension
when Unpredictable_ZEROUPPER
    return Constraint_TRUE;     // zero top halves of X registers
when Unpredictable_ERETZEROUPPERPC
    return Constraint_TRUE;     // zero top half of PC
when Unpredictable_A32FORCEALIGNPC
    return Constraint_FALSE;    // Do not force alignment
when Unpredictable_SMD
    return Constraint_UNDEF;    // disabled SMC is Unallocated
when Unpredictable_NONFAULT
    return Constraint_FALSE;    // Speculation enabled
when Unpredictable_SVEZEROUPPER
    return Constraint_TRUE;     // zero top bits of Z registers
when Unpredictable_SVELDNFDATA
    return Constraint_TRUE;     // zero top bits of Z registers
when Unpredictable_SVELDNFZERO
    return Constraint_TRUE;     // Load mem data in NF loads
when Unpredictable_AFUPDATE
    return Constraint_TRUE;     // Load mem data in NF loads
when Unpredictable_ATFUPDATE
    return Constraint_TRUE;     // Load mem data in NF loads
when Unpredictable_IESBinDebug
    return Constraint_TRUE;     // Load mem data in NF loads
when Unpredictable_SVEZEROUPPER
    return Constraint_TRUE;     // Save BTYPE in SPSR_ELx/DPSR_EL0 as '00'
when Unpredictable_CLEARERRITEZERO
    return Constraint_FALSE;    // Clearing sticky errors when instruction in flight
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

```c
// ConstrainUnpredictableBits()
// ----------------------------

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint, bits(width)) ConstrainUnpredictableBits(Unpredictable which)

    c = ConstrainUnpredictable(which);
    if c == Constraint UNKNOWN then
        return (c, Zeros(width)); // See notes; this is an example implementation only
    else
        return (c, bits(width) UNKNOWN); // bits result not used
```
// ConstrainUnpredictableBool()
// ----------------------------------

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

boolean ConstrainUnpredictableBool(Unpredictable which)
{
    c = ConstrainUnpredictable(which);
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
}

// ConstrainUnpredictableInteger()
// ---------------------------------

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.
// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which)
{
    c = ConstrainUnpredictable(which);
    if c == Constraint_UNKNOWN then
        return (c, low);                // See notes; this is an example implementation only
    else
        return (c, integer UNKNOWN);    // integer result not used
}

// General

constraint
{
Constraint.NONE, // Instruction executes with
    // no change or side-effect to its described
Constraint.UNKNOWN, // Destination register has UNKNOWN value
Constraint.UNDEF, // Instruction is UNDEFINED
Constraint.UNDEFEL0, // Instruction is UNDEFINED at EL0 only
Constraint.NOP, // Instruction executes as NOP
Constraint.TRUE,
Constraint.FALSE,
Constraint.DISABLED,
Constraint.UNCOND,
Constraint.COND,
Constraint.ADDITIONAL_DECODE, // Instruction executes with additional decode
Constraint.WBSUPPRESS, Constraint_FAULT,
    // Load-store
Constraint.FORCENOSLCHECK;
}
Library pseudocode for shared/functions/unpredictable/Unpredictable
Shared Pseudocode Functions
// Zero top 32 bits of PC on illegal return to AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// FF speculation
Unpredictable_NONFAULT,
// Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
// Load mem data in NF loads
Unpredictable_SVELDNFDATA,
// Write zeros in NF loads
Unpredictable_SVELDNFZERO,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Consider SCTLR[].IESB in Debug state
Unpredictable_IESBinDebug,
// No events selected in PMSEVFR_EL1
Unpredictable_ZEROPMSEVFR,
// No operation type selected in PMSFCR_EL1
Unpredictable_NOOPTYPES,
// Zero latency in PMSLATFR_EL1
Unpredictable_ZEROMINLATENCY,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO};
Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

```plaintext
// AdvSIMDExpandImm()
// -----------------

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
case cmode<3:1> of
  when '000'
    imm64 = Replicate(Zeros(24):imm8, 2);
  when '001'
    imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
  when '010'
    imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
  when '011'
    imm64 = Replicate(imm8:Zeros(24), 2);
  when '100'
    imm64 = Replicate(imm8:Zeros(8), 4);
  when '101'
    imm64 = Replicate(Zeros(8):imm8, 4);
  when '110'
    if cmode<0> == '0' then
      imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
    else
      imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
  when '111'
    if cmode<0> == '0' & & op == '0' then
      imm64 = Replicate(imm8, 8);
    if cmode<0> == '0' & & op == '1' then
      imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
      imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
      imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
      imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
    if cmode<0> == '1' & & op == '0' then
      imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 5):imm8<5:0>:Zeros(19);
      imm64 = Replicate(imm32, 2);
    if cmode<0> == '1' & & op == '1' then
      if UsingAArch32() then ReservedEncoding();
      imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 8):imm8<5:0>:Zeros(48);
  return imm64;
```

Library pseudocode for shared/functions/vector/PolynomialMult

```plaintext
// PolynomialMult()
// ----------------

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
result = Zeros(M+N);
extended_op2 = ZeroExtend(op2, M+N);
for i=0 to M-1
  if op1<i> == '1' then
    result = result EOR LSL(extended_op2, i);
return result;
```

Library pseudocode for shared/functions/vector/SatQ

```plaintext
// SatQ()
// ------

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
(result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
return (result, sat);
```
Library pseudocode for shared/functions/vector/SignedSatQ

```plaintext
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
if i > 2^(N-1) - 1 then
    result = 2^(N-1) - 1;  saturated = TRUE;
elsif i < -(2^(N-1)) then
    result = -(2^(N-1));  saturated = TRUE;
else
    result = i;  saturated = FALSE;
return (result<N-1:0>, saturated);
```

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```plaintext
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
assert N IN {16,32};
if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
    result = Ones(N);
else
    // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
    // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
    case N of
    when 16 estimate = RecipSqrtEstimate(UInt(operand<15:7>));
    when 32 estimate = RecipSqrtEstimate(UInt(operand<31:23>));
    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate<8:0> : Zeros(N-9);
return result;
```

Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```plaintext
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
assert N IN {16,32};
if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
    result = Ones(N);
else
    // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)
    // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
    case N of
    when 16 estimate = RecipEstimate(UInt(operand<15:7>));
    when 32 estimate = RecipEstimate(UInt(operand<31:23>));
    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate<8:0> : Zeros(N-9);
return result;
```
Library pseudocode for shared/functions/vector/UnsignedSatQ

```c
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
  if i > 2^N - 1 then
    result = 2^N - 1;  saturated = TRUE;
  elsif i < 0 then
    result = 0;  saturated = TRUE;
  else
    result = i;  saturated = FALSE;
  return (result<N-1:0>, saturated);
```

Library pseudocode for shared/translation/attrs/CombineS1S2AttrHints

```c
// CombineS1S2AttrHints()
// ======================

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc)
  MemAttrHints result;
  if HaveStage2MemAttrControl() && HCR_EL2.FWB == '1' then
    if s2desc.attrs == MemAttr_WB then
      result.attrs = s1desc.attrs;
    elsif s2desc.attrs == MemAttr_WT then
      result.attrs = MemAttr_WB;
    else
      result.attrs = MemAttr_NC;
  else
    if s2desc.attrs == '01' || s1desc.attrs == '01' then
      result.attrs = bits(2) UNKNOWN;   // Reserved
    elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
      result.attrs = MemAttr_NC;        // Non-cacheable
    elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
      result.attrs = MemAttr_WT;        // Write-through
    else
      result.attrs = MemAttr_WB;        // Write-back
  result.hints = s1desc.hints;
  result.transient = s1desc.transient;
  return result;
```
// CombineS1S2Desc()
// ===============
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)
{
    AddressDescriptor result;
    result.paddress = s2desc.paddress;

    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    elsif s2desc.memattrs.type == MemType_Device || s1desc.memattrs.type == MemType_Device then
        result.memattrs.type = MemType_Device;
        if s1desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s2desc.memattrs.device;
        else                          // Both Device
            result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device, s2desc.memattrs.device);
        end if
        result.memattrs.tagged = FALSE;
    else                        // Both Normal
        result.memattrs.type = MemType_Normal;
        result.memattrs.device = DeviceType_UNKNOWN;
        result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
        result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
        result.memattrsoutershareable = (s1desc.memattrsoutershareable || s2desc.memattrsoutershareable);
        result.memattrs.tagged = (s1desc.memattrs.tagged &&
                                result.memattrs.inner.attrs == MemAttr_WB &&
                                result.memattrs.inner.hints == MemHint_RWA &&
                                result.memattrs.outer.attrs == MemAttr_WB &&
                                result.memattrs.outer.hints == MemHint_RWA);
        result.memattrs = MemAttrDefaults(result.memattrs);
    end if

    return result;
}

// CombineS1S2Device()
// ===============
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)
{
    if s2device == DeviceType_nGnRnE || s1device == DeviceType_nGnRnE then
        result = DeviceType_nGnRnE;
    elsif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
        result = DeviceType_nGnRE;
    elsif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
        result = DeviceType_nGRE;
    else
        result = DeviceType_GRE;
    end if

    return result;
}
// LongConvertAttrsHints()
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);
    MemAttrHints result;
    if $1CacheDisabled(acctype) then // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        if attrfield<3:2> == '00' then // Write-through transient
            result.attrs = MemAttr_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elseif attrfield<3:0> == '0100' then // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
            result.transient = FALSE;
        elseif attrfield<3:2> == '01' then // Write-back transient
            result.attrs = MemAttr_WB;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        else // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;
    return result;

// MemAttrDefaults()
// Supply default values for memory attributes, including overriding the shareability attributes
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes memattrs)
    if memattrs.type == MemType_Device then
        memattrs.inner = MemAttrHints UNKNOWN;
        memattrs.outer = MemAttrHints UNKNOWN;
        memattrs.shareable = TRUE;
        memattrsoutershareable = TRUE;
    else
        memattrs.device = DeviceType UNKNOWN;
        if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
            memattrs.shareable = TRUE;
            memattrsoutershareable = TRUE;
    return memattrs;
Library pseudocode for shared/translation/attrs/S1CacheDisabled

```plaintext
// S1CacheDisabled()
// ================

boolean S1CacheDisabled(AccType acctype)
if ELUsingAArch32(S1TranslationRegime()) then
    if PSTATE.EL == EL2 then
        enable = if acctype == AccType_IFETCH then HSCTLR.I else HSCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
return enable == '0';
```

Library pseudocode for shared/translation/attrs/S2AttrDecode

```plaintext
// S2AttrDecode()
// ==============

// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)
MemoryAttributes memattrs;
apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
// Device memory
if (apply_force_writeback && attr<2> == '0') || attr<3:2> == '00' then
    memattrs.type = MemType_Device;
case attr<1:0> of
    when '00' memattrs.device = DeviceType_nGnRnE;
    when '01' memattrs.device = DeviceType_nGnRE;
    when '10' memattrs.device = DeviceType_nGRE;
    when '11' memattrs.device = DeviceType_GRE;
// Normal memory
elsif attr<1:0> != '00' then
    memattrs.type = MemType_Normal;
    if apply_force_writeback then
        memattrs.outer = S2ConvertAttrsHints(attr<1:0>, acctype);
    else
        memattrs.outer = S2ConvertAttrsHints(attr<3:2>, acctype);
        memattrs.inner = S2ConvertAttrsHints(attr<1:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    end
    memattrs = MemoryAttributes UNKNOWN; // Reserved
return MemAttrDefaults(memattrs);
```

Library pseudocode for shared/translation/attrs/S2CacheDisabled

```plaintext
// S2CacheDisabled()
// ================

boolean S2CacheDisabled(AccType acctype)
if ELUsingAArch32(EL2) then
    disable = if acctype == AccType_IFETCH then HCR2.ID else HCR2.CD;
else
    disable = if acctype == AccType_IFETCH then HCR_EL2.ID else HCR_EL2.CD;
return disable == '1';
```
Library pseudocode for shared/translation/attrs/S2ConvertAttrsHints

```plaintext
// S2ConvertAttrsHints()
// ---------------------
// Converts the attribute fields for Normal memory as used in stage 2
// descriptors to orthogonal attributes and hints

MemAttrHints S2ConvertAttrsHints(bits(2) attr, AccType acctype)
assert !IsZero(attr);
MemAttrHints result;

if S2CacheDisabled(acctype) then // Force Non-cacheable
    result.attrs = MemAttr_NC;
    result.hints = MemHint_No;
else
    case attr of
        when '01'                               // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
        when '10'                               // Write-through
            result.attrs = MemAttr_WT;
            result.hints = MemHint_RWA;
        when '11'                               // Write-back
            result.attrs = MemAttr_WB;
            result.hints = MemHint_RWA;
    end case
    result.transient = FALSE;
return result;
```

Library pseudocode for shared/translation/attrs/ShortConvertAttrsHints

```plaintext
// ShortConvertAttrsHints()
// -------------------------
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype, boolean secondstage)
MemAttrHints result;

if (!secondstage && S1CacheDisabled(acctype)) || (secondstage && S2CacheDisabled(acctype)) then // Force Non-cacheable
    result.attrs = MemAttr_NC;
    result.hints = MemHint_No;
else
    case RGN of
        when '00'                   // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
        when '01'                   // Write-back, Read and Write allocate
            result.attrs = MemAttr_WB;
            result.hints = MemHint_RWA;
        when '10'                   // Write-through, Read allocate
            result.attrs = MemAttr_WT;
            result.hints = MemHint_RA;
        when '11'                   // Write-back, Read allocate
            result.attrs = MemAttr_WB;
            result.hints = MemHint_RA;
    end case
    result.transient = FALSE;
return result;
```
Library pseudocode for shared/translation/attrs/WalkAttrDecode

```c
// WalkAttrDecode()
// ================
MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN, boolean secondstage)
{
    MemoryAttributes memattrs;
    AccType acctype = AccType_NORMAL;
    memattrs.type = MemType_Normal;
    memattrs.inner = ShortConvertAttrsHints(IRGN, acctype, secondstage);
    memattrs.outer = ShortConvertAttrsHints(ORGN, acctype, secondstage);
    memattrs.shareable = SH<1> == '1';
    memattrsoutershareable = SH == '10';
    memattrs.tagged = FALSE;
    return MemAttrDefaults(memattrs);
}
```

Library pseudocode for shared/translation/translation/HasS2Translation

```c
// HasS2Translation()
// ================
// Returns TRUE if stage 2 translation is present for the current translation regime
boolean HasS2Translation()
{
    return (EL2Enabled() && !IsInHost() && PSTATE.EL IN {EL0, EL1});
}
```

Library pseudocode for shared/translation/translation/Have16bitVMID

```c
// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.
boolean Have16bitVMID()
{
    return HaveEL(EL2) && boolean IMPLEMENTATION_DEFINED;
}
```

Library pseudocode for shared/translation/translation/PAMax

```c
// PAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the physical address size for this processor, as log2().
integer PAMax()
{
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
}
```
Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// ---------------------
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then return el;
    elif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then return EL3;
    elif HaveVirtHostExt() && ELIsInHost(el) then return EL2;
    else return EL1;

// S1TranslationRegime()
// ---------------------
// Returns the Exception level controlling the current Stage 1 translation regime. For the most part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

Library pseudocode for shared/translation/translation/VAMax

```
// VAMax()
// -------
// Returns the IMPLEMENTATION DEFINED upper limit on the virtual address size for this processor, as log2().

integer VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```