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).
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, AUTIZIB: 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.
ESB: 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, LDADDAB, 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.
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.
LDLABR: 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, LDRAAB: 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, LDSEA, 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, LDMAXA, LDMAXAL, LDMAXL: Atomic signed maximum on word or doubleword in memory.
LDMAXB, LDMAXAB, LDMAXALB, LDMAXLB: Atomic signed maximum on byte in memory.
LDMAXH, LDMAXAH, LDMAXALH, LDMAXLH: Atomic signed maximum on halfword in memory.

LDMIN, LDMINA, LDMINAL, LDMINL: Atomic signed minimum on word or doubleword in memory.
LDMINB, LDMINAB, LDMINALB, LDMINLB: Atomic signed minimum on byte in memory.
LDMINH, LDMINAH, LDMINALH, LDMINLH: 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, PACIA SP, PACIA Z, PACIA ZA: Pointer Authentication Code for Instruction address, using key A.

PACIB, PACIB1716, PACIB SP, PACIB Z, PACIB ZB: 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.

**SEF8, 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 LDADB, LDADDB, 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, LDCLRRAH, LDCLRALH, LDCLRLH.

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

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

STEORH, 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).

STLXP: Store-Release Exclusive Pair of registers.

STLXR: Store-Release Exclusive Register.

STLXRB: Store-Release Exclusive Register Byte.

STLXRH: Store-Release Exclusive Register Halfword.

STNP: Store Pair of Registers, with non-temporal hint.

STP: 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, LDSMAXL.

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, SWP AL, SWPL: Swap word or doubleword in memory.

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).


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

UBFM: Unsigned Bitfield Move.

UBFX: Unsigned Bitfield Extract: an alias of UBFM.

UDF: Permanently Undefined.

UDIV: Unsigned Divide.

UMADDL: Unsigned Multiply-Add Long.

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

UMSUBL: Unsigned Multiply-Subtract Long.

UMULL: 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.

XAFlg: 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.

ADD (scalar): Add Pair of elements (scalar).
ADD (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).

CMLT (zero): Compare signed Less 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).
FCMGT (register): Floating-point Compare Greater than (vector).
FCMGT (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).
FCMPE: 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).
FCVTAU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).
FCVTAU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).
FCVTL, FCVTL2: Floating-point Convert to higher precision Long (vector).
FCVTMS (scalar): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).
FCVTMS (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).
FCVTNU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).
FCVTNU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).
FCVTPS (scalar): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).
FCVTPS (vector): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).
FCVTPU (scalar): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).
FCVTPU (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).
FCVTZS (scalar, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).
FCVTZS (scalar, integer): Floating-point Convert to Signed integer, rounding toward Zero (scalar).
FCVTZS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTZS (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 (scalar, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).
FCVTZU (vector, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).
FDIV (scalar): Floating-point Divide (scalar).
FDIV (vector): Floating-point Divide (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).
FMAXNMV: 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).
FMINNMP (scalar): Floating-point Minimum Number of Pair of elements (scalar).
FMINNMP (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.
FRECPX: Floating-point Reciprocal exponent (scalar).
FRINT32X (scalar): Floating-point Round to 32-bit Integer, using current rounding mode (scalar).
FRINT32X (vector): Floating-point Round to 32-bit Integer, using current rounding mode (vector).
A64 -- SIMD and Floating-point Instructions (alphabetic order)

FRINT32Z (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.
LD1R: Load one single-element structure and Replicate to all lanes (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.
LD2R: Load single 2-element structure and Replicate to all lanes 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.


MOVI: 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.

RAXI: 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).
SADDLP: 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.
SM3SS1: SM3SS1.
SM3TT1A: SM3TT1A.
SM3TT1B: SM3TT1B.
SM3TT2A: SM3TT2A.
SM3TT2B: SM3TT2B.
SM4E: SM4 Encode.
SM4EKEY: SM4 Key.
SMAX: Signed Maximum (vector).
SMAXP: Signed Maximum Pairwise.
SMAXV: 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).
SQQMULH (by element): Signed saturating Doubling Multiply returning High half (by element).
SQQMULH (vector): Signed saturating Doubling Multiply returning High half.
SQQMULL, SQQMULL2 (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.

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.

SQXTN, SQXTN2: Signed saturating extract Narrow.

SQXTUN, SQXTUN2: 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).

SSHHR: 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: Signed Move vector element to general-purpose register.

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

UMULL, UMULL2 (vector): Unsigned Multiply Long (vector).

UQADD: Unsigned saturating Add.

UQRSHL: Unsigned saturating Rounding Shift Left (register).

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

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

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

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

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.

URSRA: 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.


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).
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>1</th>
<th>1</th>
<th>0</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>
<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

```java
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 = 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 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.

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;
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);
PSTATE.<N,Z,C,V> = nzcv;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.

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

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

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

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

```plaintext
integer d = UInt(Rd);
exteger n = UInt(Rn);
exteger 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>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 "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'.

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);

bit carry_in;

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

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

if d == 31 then
  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>0</th>
<th>1</th>
<th>!= 1x</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>
</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>}

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

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

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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>}
```

As an example, consider the following assembly code:

```
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

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

<table>
<thead>
<tr>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ARMv8.5)</td>
</tr>
</tbody>
</table>

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

Integer

**ADDG** <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>

```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 = 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

```python
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, uimm4, exclude);
else
    rtag = '0000';
(result, -) = if ADD then
    (result, =) = AddWithCarry(operand1, offset, '0');
result = else
    (result, =) = AddWithCarry(operand1, NOT(offset), '1');
result = AddressWithAllocationTag(result, rtag);
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Advanced SIMD

\[
\text{ADDP \langle V \rangle, \langle Vn \rangle, \langle T \rangle}
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(\text{Rd}); \\
\text{integer } n &= \text{UInt}(\text{Rn}); \\
\text{if size} &\neq '11' \text{ then UNDEFINED;} \\
\text{integer } esize &= 8 \ll \text{UInt}(\text{size}); \\
\text{integer } datasize &= esize \times 2; \\
\text{integer } elements &= 2; \\
\text{ReduceOp } op &= \text{ReduceOp ADD}.
\end{align*}
\]

Assembler Symbols

\[
\text{Is the destination width specifier, encoded in "size":}
\]

\[
\begin{array}{c|c}
\text{size} & \langle V \rangle \\
\hline
0x & \text{RESERVED} \\
10 & \text{RESERVED} \\
11 & D
\end{array}
\]

\[
\text{Is the number of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[
\text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}
\]

\[
\text{Is the source arrangement specifier, encoded in "size":}
\]

\[
\begin{array}{c|c}
\text{size} & \langle T \rangle \\
\hline
0x & \text{RESERVED} \\
10 & \text{RESERVED} \\
11 & 2D
\end{array}
\]

Operation

\[
\text{\textbf{CheckFPAdvSIMDEnabled64}}();
\]

\[
\text{\textbf{bits(datasize) operand} = } \textbf{V}[\text{n}];
\]

\[
\text{\textbf{V}[d] = Reduce(op, operand, esize);ReduceOp ADD, operand, esize);}
\]

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.
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 \(<Rm>\) 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 \texttt{CMN (extended register)}.

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

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

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

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

\begin{verbatim}
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;
\end{verbatim}

### 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.
- \(<\text{extend}>\): 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>UXTX</td>
</tr>
<tr>
<td>010</td>
<td>LSL(UXTW)</td>
</tr>
<tr>
<td>011</td>
<td>UXTW</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 <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":

<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.

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

```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, '0');

PSTATE.<N,Z,C,V> = nzcv;
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)**.

---

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

\[
\text{ADDS } <Wd>, <Wn|WSP>, \#<imm>, \langle<shift>\rangle
\]

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

\[
\text{ADDS } <Xd>, <Xn|SP>, \#<imm>, \langle<shift>\rangle
\]

```plaintext
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;

\[\text{case shift of}
\begin{align*}
\text{when '00' imm} & \text{ = ZeroExtend(imm12, datasize);} \\
\text{when '01' imm} & \text{ = ZeroExtend(imm12, Zeros(12), datasize);} \\
\text{when '10'} & \text{ SEE "ADDG, SUBG";} \\
\text{when '11' } & \text{ ReservedValue();}
\end{align*}
\]
```

**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>CMN (immediate)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

```c
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, imm, '0');
(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = nzcv;
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)**.

32-bit (sf == 0)

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

64-bit (sf == 1)

ADDS <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.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (shifted register)</td>
<td>Rd == '111111'</td>
</tr>
</tbody>
</table>
Operation

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, '0'};
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.

Advanced SIMD

ADDV <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; Rd = ReduceOp_ADD(operand, esize);

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];
V[d] = ReduceOp_ADD(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.

(htmldiff from- (old) (new))
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   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | immlo| 1   | 0   | 0   | 0   | immhi|       | Rd  |

**Literal**

ADR `<Xd>, <label>`

```plaintext
type d = UInt(Rd);
bool page = (op == '1');
bits(64) imm;

imm = if page then
          signExtend(immhi:immlo, 64);
       else
          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 address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

**Operation**

```plaintext
bits(64) base = PC[11:0];

if page then
  base[11:0] = zeros(12);

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

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:25:04
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 0  | 0  | 0  | immlo | | immhi | | Rd | | op |
```

**Literal**

ADRP <Xd>, <label>

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

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

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

X[d] = base + imm;
```
AESD

AES single round decryption.

```
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 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  |  0  |  0  |  0  |  1  |  0  |  0  |  0  |  0  |  1  |  0  |  1  |  0  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

Advanced SIMD

AESD <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 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;
result = 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.
AESE

AES single round encryption.

Advanced SIMD

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

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
if 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;
result = if decrypt then
        AESInvSubBytes(AESInvShiftRows(result));
else
        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.

<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>0</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>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>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>

Advanced SIMD

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

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;

boolean decrypt = (D == '1')

D
```

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;
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.
AESMC

AES mix columns.

```
<table>
<thead>
<tr>
<th>D</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Advanced SIMD

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

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;

boolean decrypt = (D == '1');
```

Assemble 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

```python
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n];
bits(128) result;
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 1  | Rm | 0  | 0  | 0  | 1  | 1  | Rn | 1  | 1  | Rd | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  |

Three registers of the same type

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

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

integer datasize = if Q == '1' then 128 else 64;
integer esize = 8;
integer elements = datasize DIV esize;

integer 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

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = operand1 AND operand2;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.
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.

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

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

64-bit (sf == 1)

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

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bite(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = 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;
result = operand1 AND imm;
if d == 31 then 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>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)

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;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

boolean setflags;

enum LogicalOp =
  LogicalOp_AND;
  LogicalOp_ORR;
  LogicalOp_EOR;
  LogicalOp_AND;

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);
result = operand1 AND operand2; if invert then operand2 = NOT(operand2);
```

```plaintext
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 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.

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

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

ANDS <Wd>, <Wn>, #<imm>

64-bit (sf == 1)

ANDS <Xd>, <Xn>, #<imm>

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

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;
result = operand1 AND imm;
PSTATE.<N,Z,C,V> = result<datasize-1>:case_op_of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_OR operand1 OR operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;
if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:sZeroBit('00');
if d == 31 || !setflags then
    X[d] = 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).

32-bit (sf == 0)

\[
\text{ANDS <Wd>, <Wn>, <Wm>}, (<shift> \#<amount>)
\]

64-bit (sf == 1)

\[
\text{ANDS <Xd>, <Xn>, <Xm>}, (<shift> \#<amount>)
\]

```haskell
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
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);\#<amount>;
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

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>; if invert then operand2 = NOT(operand2);

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

  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.
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  | 1  | 0  | 1  | 1  | 1  | 1  |

CRm

System

AXFlag

if !HaveFlagFormatExt() then UNDEFINED;

Operation

\[
\begin{align*}
\text{bit } N &= '0'; \\
\text{bit } Z &= \text{PSTATE}.Z \text{ OR } \text{PSTATE}.V; \\
\text{bit } C &= \text{PSTATE}.C \text{ AND NOT}(\text{PSTATE}.V); \\
\text{bit } V &= '0'; \\
\text{PSTATE}.N &= N; \\
\text{PSTATE}.Z &= Z; \\
\text{PSTATE}.C &= C; \\
\text{PSTATE}.V &= V; \\
\end{align*}
\]

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.

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

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(cond) then
  BranchTo(PC[] + offset, BranchType_DIR);

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.

<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>0</td>
<td>1</td>
<td>imm26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

26-bit signed PC-relative branch offset

```plaintext
B <label>

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

```plaintext
if branch_type == BranchType_DIRCALL then X[30] = PC[] + 4;

BranchTo(PC[] + offset, BranchType_DIR);
```

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

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

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 \textbf{BFC}, \textbf{BFI}, and \textbf{BFXIL}.

<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>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>
</tr>
</tbody>
</table>

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

\[ \text{BFM }<Wd>, \langle Wn>, \#, <\text{immr}>, \#, <\text{imms}> \]

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

\[ \text{BFM }<Xd>, \langle Xn>, \#, <\text{immr}>, \#, <\text{imms}> \]

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(\text{datasize} = \text{if } sf == '1' \text{ then } 64 \text{ else } 32\);

Boolean \(\text{inzero}\);
Boolean \(\text{extend}\);
integer \(R\);
integer \(S\);
bits(\(\text{datasize}\)) \(wmask\);
bits(\(\text{datasize}\)) \(tmask\);

\text{case } opc \text{ of}
\text{ when } '00' \text{ inzero } = \text{TRUE}; \text{ extend } = \text{TRUE}; \text{ // SBFM}
\text{ when } '01' \text{ inzero } = \text{FALSE}; \text{ extend } = \text{FALSE}; \text{ // BFM}
\text{ when } '10' \text{ inzero } = \text{TRUE}; \text{ extend } = \text{FALSE}; \text{ // UBFM}
\text{ when } '11' \text{ UNDEFINED;}

\text{if } sf == '1' \&\& N != '1' \text{ then } \text{UNDEFINED;}
\text{if } sf == '0' \&\& (N != '0' \text{ || immr<5> != '0' \text{ || imms<5> != '0'}) \text{ then } \text{UNDEFINED;}

\(R = \text{UInt}(\text{immr});\)
\((wmask, tmask) = S = \text{UInt}(\text{imms});\)
\((wmask, tmask) = \text{DecodeBitMasks}(N, \text{imms}, \text{immr}, \text{FALSE});\)

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.
\(<\text{immr}>\) For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.
\(<\text{imms}>\) For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
\(<\text{imms}>\) 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; (\text{UInt}(\text{imms}) &lt; \text{UInt}(\text{immr}))</td>
</tr>
<tr>
<td>BFI</td>
<td>Rn != '11111' &amp;&amp; (\text{UInt}(\text{imms}) &lt; \text{UInt}(\text{immr}))</td>
</tr>
<tr>
<td>BFXIL</td>
<td>(\text{UInt}(\text{imms}) \geq \text{UInt}(\text{immr}))</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
\text{bits}(\text{datasize}) \text{ dst} = \text{if \text{inzero} then \text{zeros}(d) else \text{X}[d]};
\text{bits}(\text{datasize}) \text{ src} = \text{X}[n];

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

// determine extension bits (sign, zero or dest register)
\text{bits}(\text{datasize}) \text{ top} = \text{if \text{extend} then \text{Replicate}(\text{src}, R) AND \text{wmask})};
\text{if}(\text{src}<\text{d}) \text{ else} \text{dst};

// combine extension bits and result bits
\text{X}[d] = (\text{dst AND NOT}(\text{tmask})) OR (\text{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.
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.

16-bit (cmode == 10x1)

BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}

32-bit (cmode == 0x1)

BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

AsmImmediateOp operation;

AsmCase cmode:op of
  when '0xx01' operation = ImmediateOp_MOVI;
  when '0xx00' operation = ImmediateOp_MVNI;
  when '0xx11' operation = ImmediateOp_BIC;
  when '0xx10' operation = ImmediateOp_ORR;
  when '10x01' operation = ImmediateOp_MOVI;
  when '10x00' operation = ImmediateOp_MVNI;
  when '10x11' operation = ImmediateOp_BIC;
  when '10x10' operation = ImmediateOp_ORR;
  when '110x1' operation = ImmediateOp_MOVI;
  when '110x0' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '1111x' operation = ImmediateOp_MOVI;

immediate operation;

AsmCase Q:op of
  when 0 operation = ImmediateOp_MOVI;
  when 1 operation = ImmediateOp_MVNI;

imm64 = AsvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = AsmReplicate(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>
Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

For the 16-bit 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>0</td>
</tr>
</tbody>
</table>

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**

```
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 1  | 1  | 1  | 1  | Rm | 0  | 0  | 0  | 1  | 1  | 1  | Rn | Rd |

Three registers of the same type

BIC <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 “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 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);
result = operand1 AND 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>1</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>
<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)

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

64-bit (sf == 1)

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

```cpp
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
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);
operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);
result = operand1 AND operand2; case op of
    LogicalOp_AND result = operand1 AND operand2;
    LogicalOp_ORR result = operand1 OR operand2;
    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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | 0  | 0  | 1  | Rm | imm6| Rn | Rd |
| opc |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

if sf == '0' && imm6<5> == '1' then UNDEFINED; 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 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);

result = operand1 AND operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:
        case op of
            LogicalOp_AND result = operand1 AND operand2;
            LogicalOp_ORR result = operand1 OR operand2;
            LogicalOp_EOR result = operand1 EOR operand2;

    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 \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{center}
\begin{tabular}{ccccccccccccccccccc}
\hline
0 & Q & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & Rm & 0 & 0 & 0 & 1 & 1 & 1 & Rn & 0 & 0 & 0 & 1 & 1 & Rd & 0 & 0 & 0 & 1 & 1 & 1
\end{tabular}
\end{center}

\textbf{Three registers of the same type}

BIF \texttt{<Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

\begin{Verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8;
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;
\end{Verbatim}

\textbf{Assembler Symbols}

\begin{itemize}
  \item \texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the “Rd” field.
  \item \texttt{<T>} Is an arrangement specifier, encoded in “Q”:
  \begin{tabular}{|c|c|}
    \hline
    Q & \texttt{<T>} \\
    \hline
    0 & 8B \\
    1 & 16B \\
    \hline
  \end{tabular}
  \item \texttt{<Vn>} Is the name of the first SIMD&FP source register, encoded in the “Rn” field.
  \item \texttt{<Vm>} Is the name of the second SIMD&FP source register, encoded in the “Rm” field.
\end{itemize}
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = [n];

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

V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3); 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.
**BIT**

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 *CPACR_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  | 1  | 0  | 1  | Rm | 0  | 0  | 0  | 1  | 1  | Rn | 1  | Rd |

### Three registers of the same type

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

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8;
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;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];
operand1 = case op of
    when VBitOp_VEOR
        operand1 = V[m];
        operand2 = Zeroes();
        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];
        operand3 = operand2 = operand1;
        operand3 = NOT(V[m]);
    V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);
    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.

![](image)

BL

26-bit signed PC-relative branch offset

BL <label>

```assembly
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

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

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

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

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+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  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Integer

BLR <Xn>

```python
integer n = UInt(Rn); BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_m = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UNDEFINED;
elsif pac && !HavePACExt() then
    UNDEFINED;
endcase 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.
```c
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';
    end;

  if pac then
    bits(64) modifier = if source_is_sp then SP[] else X[30] = m;
    if use_key_a then
      target = AuthIA(target, modifier);
    else
      target = AuthIB(target, modifier);
    end;
  if branch_type == BranchType_INDCALL then X[30] = PC[] + 4;
  BranchTo(target, BranchType_INDCALL);
```

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

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 <Xn>, 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 <Xm|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.

**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  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | M  | Rn |   |   |   |   |   |   |   |   |

op    A
```
Key A, zero modifier (Z == 0 && M == 0 && Rm == 11111)

BLRAAZ <Xn>

Key A, register modifier (Z == 1 && M == 0)

BLRAA <Xn>, <Xm|SP>

Key B, zero modifier (Z == 0 && M == 1 && Rm == 11111)

BLRABZ <Xn>

Key B, register modifier (Z == 1 && M == 1)

BLRAB <Xn>, <Xm|SP>

```c
integer n = UInt(Rn);
integer m = UInt(Rm); branch_type = 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;
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
    UNDEFINED;
  if Z == '0' && m != 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.

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.
bits(64) target = X[n];
bits(64) modifier = if source_is_sp then 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, BranchType_INDCALL);(target, branch_type);
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  | 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  |
```

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;
else if 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

\[
\text{bits}(64) \text{ target} = X[n];
\]

\[
\text{case branch type of}
\]

\[\text{BranchType_INDIR}\]

\[\text{if InGuardedPage then}
\]

\[\text{if } n = 16 \text{ or } n = 17 \text{ then}
\]

\[\text{BTypeNext} = '01';
\]

\[\text{else}
\]

\[\text{BTypeNext} = '11';
\]

\[\text{else}
\]

\[\text{BTypeNext} = '01';
\]

\[\text{when BranchType_INDCALL}
\]

\[\text{BTypeNext} = '10';
\]

\[\text{when BranchType_RET}
\]

\[\text{BTypeNext} = '00';
\]

\[\text{if pac then}
\]

\[\text{bits}(64) \text{ modifier} = \text{if source is sp then } SP \text{ else } X[m];
\]

\[\text{if use key a then}
\]

\[\text{target} = \text{AuthIA}(\text{target}, \text{modifier});
\]

\[\text{else}
\]

\[\text{target} = \text{AuthIB}(\text{target}, \text{modifier});
\]

\[\text{if branch type = BranchType_INDCALL then } X[30] = PC[] + 4;
\]

\[\text{Branch?c}(\text{target, BranchType_INDIR});\]

\[\text{BranchTo}(\text{target, branch type});
\]
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)

```
  31  30  29  28  27  26  25  24  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  Z  0  0  1  1  1  1  0  0  0  1  M  Rn  Rm

  op  A
```
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>

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer p = 1 == '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;
  end;
  if pac then
    if Z == '0' && m != 31 then
      UNDEFINED;
      if branch_type == BranchType_RET() then
        UNDEFINED;
      if Z == '0' && m != 31 then
        UNDEFINED;
        if n != 31 then UNDEFINED;
        n = 30;
        source_is_sp = TRUE;
    end;
  end;
end;
```

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

bits(64) target = X[n];
bits(64) modifier = if source_is_sp then case branch_type of
    BranchType_INDIR:
        if InGuardedPage then
            if n == 16 || n == 17 then
                BTypeNext = '01';
            else
                BTypeNext = '11';
            else
                BTypeNext = '01';
    BranchType_INDCALL:
        BTypeNext = '10';
    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, BranchType_INDIR);(target, branch_type);

(old) htmldiff from- (new)
BRK

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.

| Bit 31 | Bit 30 | Bit 29 | Bit 28 | Bit 27 | Bit 26 | Bit 25 | Bit 24 | Bit 23 | Bit 22 | Bit 21 | Bit 20 | Bit 19 | Bit 18 | Bit 17 | Bit 16 | Bit 15 | Bit 14 | Bit 13 | Bit 12 | Bit 11 | Bit 10 | Bit 9 | Bit 8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 1      | 1      | 0      | 1      | 0      | 0      | 0      | 0      | 1      | 0      | 0      | 0      | 0      | 0      | 0      | 0      | 0      | 0      | 0      | 0      | 0      | imm16  | 0      | 0      | 0      | 0      | 0      |

System

BRK #<imm>

```plaintext
= @bits(16) comment = imm16;
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

```plaintext
AArch64.SoftwareBreakpoint(imm16);4(comment);
```
**BSL**

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.

<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>Q</td>
</tr>
</tbody>
</table>

**Three registers of the same type**

BSL `<Vd>..<T>`, `<Vn>..<T>`, `<Vm>..<T>

```plaintext
d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8;
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”:
  - | Q | `<T>` |
  - | 0 | 8B |
  - | 1 | 16B |
- `<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
void CheckFPAdvSIMDEnabled64()
{
    bits(datasize) operand1;
    bits(datasize) operand2;
    bits(datasize) operand3;
    bits(datasize) operand4 = V[n];

    operand1 = 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[m];
        operand3 = operand1
            AND NOT(V[d] ^ V[m]);
    V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);
    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 != 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 <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 != 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)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | x  | x  | 0  | 1  | 1  | 1  | 1  |     |
```

CRm | op2

System

BTI {<targets>}

```java
SystemHintOp op;
if CRm:op2 == '0100 xx0' then
  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 "PACIALI, PACIALI6, AUTIALI6, AUTIBI6";
    when '0010 000' << HAVEPERF() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_TSB;
    when '0010 001' << HAVEStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
    when '0010 010' << HAVESelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_CSDB;
    when '0011 xxx' SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    when '0100 xx0' == SystemHintOp_BTI;
    otherwise BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
else
  EndOfInstruction(); // Instruction executes as NOP
```
Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;targets&gt;</th>
<th>&lt;op2&lt;2:1&gt;&gt;</th>
<th>&lt;targets&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>jc</td>
<td></td>
</tr>
</tbody>
</table>

Operation

case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      CleanEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFETraps(EL1, TRUE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFETraps(EL2, TRUE);
      if HaveEL(EL2) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFETraps(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.CheckForWFETraps(EL1, FALSE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFETraps(EL2, FALSE);
      if HaveEL(EL2) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFETraps(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();
    TakeUnmaskedErrorInterruption();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  otherwise  // do nothing
    when SystemHintOp_BTI
      BTypeNext = '00';
  otherwise  // do nothing
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.

- CASAB 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  | 0  | 0  | 0  | 1  | 0  | 1  | L | 1 | Rs | o0 | 1 | 1 | 1 | 1 | 1 | Rn | 0  | 0  | 0  | 0  | 1 | 0  | 0  | 0  | 0  | 1 | L | 1 | Rs | o0 | 1 | 1 | 1 | 1 | 1 | Rn | 0  | 0  | 0  | 0  | 1 | 0  | 0  | 0  |

size

CASB (L == 1 && o0 == 0)

CASAB <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.
Operation

bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;
bits(8) data;
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
    CheckSPEAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];
data, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, 1, stacctype] = newvalue;
data, datasize DIV 8, stacctype] = newvalue;
X[s] = ZeroExtend(data, 32);(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.

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 Rn  
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 \(<Ws>\), \(<Wt>\), \([<Xn|SP>{{#0}}]\)
```

```java
if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
integer datasize = 8 <<
      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(16) comparevalue;
bits(16) newvalue;
bits(16) data;
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, 2, ldacctype];
    address, datasize DIV 8, ldacctype];

    if data == comparevalue then
        Mem[address, 2, stacctype] = newvalue;
        Mem[address, datasize DIV 8, stacctype] = newvalue;

    X[s] = ZeroExtend(data, 32); //data, regsize);


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](#).

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)

```
0 sz 0 1 0 0 0 0 0 L 1 Rs 0 1 1 1 1 1 Rn Rt
```

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\}]

```plaintext
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)4sz;
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.
Operation

```c
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();
    address = SP[];
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>, datasize);
    X[s+1] = ZeroExtend(data<datasize-1:0>, datasize);
else
    X[s] = ZeroExtend(data<datasize-1:0>, datasize);
    X[s+1] = ZeroExtend(data<2*datasize-1:datasize>, datasize);
```

*<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.

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

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

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>`: 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

```
bits(datasize) operand1 = X[t];
if IsZero(operand1) == FALSE then iszero then BranchTo(PC[] + offset, BranchType_DIR);
```

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.

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

CBZ <Wt>, <label>

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

CBZ <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>` 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**

```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == TRUE then iszero then
    BranchTo(PC[] + offset, BranchType_DIR);
```

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

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.

32-bit (sf == 0)

CCMN <Wn>, #<imm>, #<nzcv>, <cond>

64-bit (sf == 1)

CCMN <Xn>, #<imm>, #<nzcv>, <cond>

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

Operation information

If PSTATE.DIT is 1:

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

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(cond) then
  if sub_op then
    operand2 = NOT(operand2);
  carry_in = '1';
  (-, flags) = AddWithCarry(operand1, operand2, '0');
  (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.

|   |   |   |   |   |   | 1 | 0 | Rn |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|
|sf| 1| 1| 1| 1| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|
|op|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

32-bit (sf == 0)

CCMP <Wn>, #<imm>, #<nzcv>, <cond>

64-bit (sf == 1)

CCMP <Xn>, #<imm>, #<nzcv>, <cond>

integer n = UInt(Rn);
integer datadsize = is sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
bits(datadsize) imm = ZeroExtend(imm5, datadsize);

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(dataradsize) operand1 = X[n];
bite(dataradsize) operand2;
bite(dataradsize) operand2 = imm;
bite carry_in = '0';
if ConditionHolds(condition) then
  operand2 = NOT(imm);
if sub_op then
  operand2 = NOT(operand2);
carry_in = '1';
(-, flags) = AddWithCarry(operand1, operand2, '1');
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.

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

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';
if ConditionHolds(cond) then
    operand2 = NOT(operand2);
else if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
(operand1, operand2, carry_in) = AddWithCarry(operand1, operand2, '1');
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.
CLS

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

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;
integer 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];
result = if opcode == CountOp_CLS 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.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 | 0 | 0 | 1 | 0 | 0 | Rn | Rd
```

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

```
CLZ <Wd>, <Wn>
```

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

```
CLZ <Xd>, <Xn>
```

```plaintext
int d = UInt(Rd);
int n = UInt(Rn);
int 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
int result;
bits(datasize) operand1 = X[n];
result = if opcode == CountOp_CLZ then
          CountLeadingZeroBits(operand1);
else
          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.
CRC32B, CRC32H, CRC32W, CRC32X

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. ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

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

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>

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.

If !HaveCRCEx() 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');

Page 114
Operation

```
bits(32) acc = X[n];  // accumulator
bits(size) val = X[m];  // input value
bits(32) poly = 0x04C11DB7<31:0>; // input value
bits(32) poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;

bits(32+size) tempacc = BitReverse(acc);(acc) : Zeros(size);
bits(size+32) tempval = BitReverse(val);(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.
CRC32CB, CRC32CH, CRC32CW, CRC32CX

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.

31 30 29 28 27 26 25 24 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 | Rm | 0 | 1 | 0 | 1 | sz | Rn | Rd |

CRC32CB (sf == 0 && sz == 00)

CRC32CH (sf == 0 && sz == 01)

CRC32CW (sf == 0 && sz == 10)

CRC32CX (sf == 1 && sz == 11)

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 = 0x1EDC6F41<31:0>; // 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.

(old) htmldiff from- (new)
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.

```cpp
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_CSDB;
  when '0001 xxx' SEE "XPACLRI";
  when '0010 000' if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
  when '0010 001' if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
  when '0010 010' if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
  when '0010 100' op = SystemHintOp_BTI;
  when '0010 101' op = SystemHintOp_ESB;
  when '0011 xxx' SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0' BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction(); // Empty.
```

case op of
  when SystemHintOp_YIELD();
  when SystemHintOp_WFE
    if !IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE_EL == $10 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFPxTrap($10, TRUE);
      if EL2Enabled() && PSTATE_EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFPxTrap($12, TRUE);
      if HaveEL($13) && PSTATE_EL == $13 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFPxTrap($13, TRUE);
      WaitForEvent();
  when SystemHintOp_WFI
    if !InterruptPending() then
      if PSTATE_EL == $10 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFPxTrap($10, FALSE);
      if EL2Enabled() && PSTATE_EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFPxTrap($12, FALSE);
      if HaveEL($13) && PSTATE_EL == $13 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFPxTrap($13, FALSE);
      WaitForEvent();
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
    if EL2Enabled() && PSTATE_EL IN {EL0, EL1, EL2} then
      AArch64.vESBOperation();
    TakeUnmaskedErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI();
    BTypeNext = '00';
  otherwise // do nothing
CSEL

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.

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

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

if ConditionHolds(cond) 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.
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>

```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>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

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

if ConditionHolds(cond) then
result = operand1;
else
result = operand2 + 1;
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.

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

<table>
<thead>
<tr>
<th>sf</th>
<th>1 0 1 1 0 1 0 0 Rm</th>
<th>cond</th>
<th>0 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CSINV <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSINV <Xd>, <Xn>, <Xm>, <cond>

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

integer datasize = if sf == '1' then 64 else 32;
integer 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>
```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds[cond] then
  result = operand1;
else
  result = NOT(operand2);

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.

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

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 != '111x' &amp; Rn == Rm</td>
</tr>
</tbody>
</table>
Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    result = operand1;
else
    result = NOT(operand2);
    result = result + 1;
    result = operand2;
else_inv then result = NOT(result);
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.
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.

```
System

DCPS1 {#<imm>}

   bits(2) target_level = LL;
   if LL == '00' then UNDEFINED;
   if !Halted() then AArch64.UndefinedFault();
```

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(LL);(target_level);
```

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 0 | 1 0 1 | imm16 | 0 0 0 | 1 0

System

DCPS2 {#<imm>}

```

```
31 30 29 28 27 26 25 24 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 | 1 0

Asm

```

Asm

```

```

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

```

```

```

DCPS2
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 $UNKNOWN$.
- $DLR_{EL0}$ and $DSPSR_{EL0}$ become $UNKNOWN$.
- The endianness is set according to $SCTLR_{EL3}.EE$.

This instruction is $UNDEFINED$ at all exception levels if either:

- $EDSCR.SDD == 1$
- EL3 is not implemented.

This instruction is always $UNDEFINED$ in Non-debug state.

For more information on the operation of the DCPSn instructions, see $DCPS$.

```
31 30 29 28 27 26 25 24 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 | 0 0 1 1
    imm16
          ^
LL
```

System

```
DCPS3 {#<imm>}

bit(2) target_level = LL;
if LL == '00' then UNDEFINED;
if !Halted() then AArch64.UndefinedFault();
```

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(LL);(target_level);
```

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

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

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  |

opc

CRm

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**

```pseudo
case op of
  when MemBarrierOp_DSBDataSynchronizationBarrier (domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier (domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier () ;
  when MemBarrierOp_SSBSpeculativeSynchronizationBarrierToVA () ;
  when MemBarrierOp_PSSBSpeculativeSynchronizationBarrierToPA () ;
  when MemBarrierOp_SBSpeculationBarrier () ;
```

(old) htmldiff from-  (new)
Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see Data Synchronization Barrier.

System

```assembly
    DSB <option>|#<imm>

    MemBarrierOp op;
    MBReqDomain domain;
    MBReqTypes types;

    op = case opc of
        when '00' op = MemBarrierOp_DSB;
    case CRm<3:2> of
        when '00' domain = 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' then
                if HaveSBExt() && FALSE then
                    op = MemBarrierOp_SB;
                else
                    types = MBReqTypes_All;
                    domain = MBReqDomain_FullSystem;
            else
                op = MemBarrierOp_SSBB;
```

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_DSBDDataSynchronizationBarrier (domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier (domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier();
  when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
  when MemBarrierOp_SBSpeculationBarrier();
```

(htmldiff from- (new))
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.

<table>
<thead>
<tr>
<th>sf</th>
<th>1 0 0 0 1 0 1 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)

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

64-bit (sf == 1)

EON <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;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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

\[ \text{bits}(\text{datasize}) \text{ operand1} = X[n]; \]
\[ \text{bits}(\text{datasize}) \text{ operand2} = \text{ShiftReg}(m, \text{shift\_type}, \text{shift\_amount}); \]
\[ \text{operand2} = \text{NOT(operand2)}; \]
\[ \text{if \ invert \ then \ operand2} = \text{NOT(operand2)}; \]
\[ \text{result} = \text{operand1} \text{ EOR operand2}; \]
\[ \text{case \ op \ of} \]
\[ \text{when} \]
\[ \text{logical\_op\_AND} \] \[ \text{result} = \text{operand1} \text{ AND operand2}; \]
\[ \text{when} \]
\[ \text{logical\_op\_ORR} \]
\[ \text{result} = \text{operand1} \text{ OR operand2}; \]
\[ \text{when} \]
\[ \text{logical\_op\_EOR} \]
\[ \text{result} = \text{operand1} \text{ EOR operand2}; \]
\[ \text{if \ setflags \ then} \]
\[ \text{PSTATE.}CN,Z,C,V = \text{result}\text{<datasize-1>:} \text{IsZeroBit(result):'00'}; \]
\[ X[d] = \text{result}; \]

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 (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>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>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

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

```plaintext
d = Int(Rd);
n = Int(Rn);
m = Int(Rm);
dataSize = if Q == '1' then 128 else 64;
esize = 8;
ed_elements = datasize DIV esize;

opc2 = case opc2 of
   '00' => VBitop_VEOR;
   '01' => VBitop_VBSL;
   '10' => VBitop_VBIT;
   '11' => VBitop_VBIF;
end;
```

### 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;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = 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.

(old)htmldiff from- (new)
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 1  | 0  | 0  | N  | immr | imms | Rn | Rd |

opc

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;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = boolean setflags; LogicalOp op
case opc of
   when '00' op = LogicalOp_AND, setflags = FALSE;
   when '01' op = LogicalOp_OR, setflags = FALSE;
   when '10' op = LogicalOp_AND, setflags = FALSE;
   when '11' op = LogicalOp_AND, setflags = TRUE;
endcase;
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;
result = operand1 EOR imm;
if d == 31 then
    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.

(old)  htmldiff from-  (new)
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.

\[
\begin{array}{cccccccccccccccccc}
\hline
sf & 1 & 0 & 0 & 1 & 0 & 1 & 0 & shift & 0 & Rm & imm6 & Rn & Rd & 0 & 0 & 0 & 1 & 0 & 1 & 0 & N & opc
\end{array}
\]

32-bit (sf == 0)

EOR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}</code>

64-bit (sf == 1)

EOR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}</code>

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED; 
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>

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);
result = operand1 EOR operand2; 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 \textit{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 \textit{Illegal return events from AArch64 state}.

\textbf{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  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

\textbf{System}

\textbf{ERET}

```
if PSTATE.EL == EL0 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;
```

\textbf{Operation}

```
AArch64.CheckForERetTrap(FALSE, TRUE);
(pac, use_key_a);
bits(64) target = ELR[];
if pac then
  if use_key_a then
    target =
AuthIA(ELR[], SP[]);
  else
    target = AuthIA(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 PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for ERETAA, and key B is used for 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 Illegal return events from AArch64 state.

ERETAA and ERETAB are 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  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |

ERETAA (M == 0)

ERETAA

ERETAB (M == 1)

ERETAB

if PSTATE.EL == EL0 then UNDEFINED;

boolean pac = (A == '1');

boolean use_key_a = (M == '0');

if !pac && op4 != '00000' then

UNDEFINED;

elsif pac && (!HavePACExt() then

UNDEFINED;

if Rn != '11111' then

UNDEFINED;

Operation

AArch64.CheckForERetTrap(TRUE, use_key_a);

bits(64) target;

if use_key_a then

    target = [pac, use_key_a];

else

    if pac then

        if use_key_a then

            target = AuthIA(ELR[], SP[]);

        else

            target = AuthIB(ELR[], SP[]);

    else

        target = 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)

31 30 29 28 27 26 25 24 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 1 0 0 0 1 0 0 0 1 1 1 1

CRm op2

System

ESB

```c
System HintOp if !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 '0011 xxx'
            SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
        when '0100 xx0'
            if BTypeCompatible = BTypeCompatible_BTI() then (op2<2:1>);
            op = SystemHintOp_BTI;
        otherwise
            EndOfInstruction();(); // Instruction executes as NOP
    
    
```

ESB
Operation

```c
case op of
  when SystemHintOp_YIELD() then
  hint_Yield();
  when SystemHintOp_WFE then
    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); // Check for traps described by the Hypervisor.
      if EL2Enabled() && PSTATE.EL IN {EL1, EL2} && !IsInHost() then
        AArch64.CheckForWFETrap(EL2, TRUE); // Check for traps described by the Secure Monitor.
      if HaveEL(EL3) && PSTATE.EL == EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFETrap(EL3, TRUE);
        WaitForEvent();
      when SystemHintOp_WFI then
        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); // Check for traps described by the Hypervisor.
            if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
              AArch64.CheckForWFETrap(EL2, FALSE); // Check for traps described by the Secure Monitor.
            if HaveEL(EL3) && PSTATE.EL == EL3 then
              // Check for traps described by the Secure Monitor.
              AArch64.CheckForWFETrap(EL3, FALSE);
              WaitForInterrupt();
            when SystemHintOp_SEV then
              SendEvent();
            when SystemHintOp_SEVL then
              SendEventLocal();
            when SystemHintOp_ESB then
              SynchronizeErrors();
              AArch64.ESBOperation();
              if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
                AArch64.vESBOperation();
                TakeUnmaskedSErrorInterrupts();
            when SystemHintOp_PSB then
              ProfilingSynchronizationBarrier();
            when SystemHintOp_TSB then
              TraceSynchronizationBarrier();
            when SystemHintOp_CSDB then
              ConsumptionOfSpeculativeDataBarrier();
            when SystemHintOp_BTI() then
              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.

Depending on the settings in the CPACR_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

EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>

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

if Q == '0' & &imm4<3> == '1' then UNDEFINED;

integer datasize = if Q == '1' then 128 else 64;
integer position = 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.
<br> 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

CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m];
bits(datasize) lo = V[n];
bits(datasize*2) concat = hi:lo;
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.
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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>type</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
</table>
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | opc

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
data size = 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
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = 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.
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   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

<table>
<thead>
<tr>
<th>type</th>
<th>Rm</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

op
```

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>
```

```java
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;
        end
    end
boolean sub_op = (op == '1');
```
Operation

```c
CheckFPAdvSIMDEnabled64();

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

result = 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: 2018-09-13T13:25:04

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  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  |
| sz |

Half-precision

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

if !HaveFP16Ext () then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = 32; if sz == '1' then UNDEFINED;
integer datasize = esize * 2;
integer elements = 2; ReduceOp op = ReduceOp_FADD;

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  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  |

Single-precision and double-precision

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

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

integer esize = 32;
integer datasize = 64; integer esize = 32; if esize != 32 then UNDEFINED;
integer datasize = esize * 2;
integer elements = 2; ReduceOp op = ReduceOp_FADD;

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</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 half-precision variant: is the destination width specifier, encoded in “sz”:

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);ReduceOp_FADD, operand, esize);
```

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

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

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

(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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | Rm | 1  | 1  | 1  | rot | 0  | 1  | Rn | Rd |

Three registers of the same type

FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

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

```c
CheckFPAdvSIMEnabled64();

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
    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: 2018-09-13T13:25:04

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 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. 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 on the CPACR_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
|   |   |   |   | type |   |   | Rm |   |   | cond |   |   | Rn |   |   | nzcv | op |
```

**Half-precision (type == 11)**

(ARMv8.2)

```
FCCMP <Hn>, <Hm>, #<nzcv>, <cond>
```

**Single-precision (type == 00)**

```
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
```

**Double-precision (type == 01)**

```
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
```

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;

**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.
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
CheckFPAdvSIMEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];

if ConditionHolds[cond] then
    flags = FPCompare(operand1, operand2, FALSE, FPCR);
    PSTATE.<N,Z,C,V> = flags;
```

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

### 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
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
datase = 16;
    else
      UNDEFINED;

  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.
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 \(<\), \(\leq\), \(>\), \(\geq\), and other predicates that raise an exception when the operands are unordered.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];
if ConditionHolds(cond) then
    flags = FPCompare(operand1, operand2, TRUE, FPCR);
    PSTATE.<N,Z,C,V> = flags;
```

```
(old) htmldiff from- (new)
```
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.

**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”:

```assembly
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;
integer datasize = 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;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q == '0' then UNDEFINED;
```
<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

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to (elements DIV 2)-1
  case rot of
  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+1, esize];
      when '10'
        element1 = FPNeg(Elem[operand2, index*2, esize]);
        element2 = Elem[operand1, e*2, esize];
        element3 = FPNeg(Elem[operand2, index*2+1, 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+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;
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 | 1 | 0 | size | 0 | Rm | 1 | 1 | 0 | rot | 1 | Rd |

Three registers of the same type

FCMLA <Vd>..<T>, <Vn>..<T>, <Vm>..<T>, #<rotate>

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

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
  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);
  end case
end for
V[d] = result;
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  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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>

```plaintext
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;
            UNDEFINED;
    end
end

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

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;

result = if ConditionHolds(cond) 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.
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.

```
31 30 29 28 27 26 25 24 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 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
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;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if integer fltsize; FPConv_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
          fltsize = 128;
        else
          fltsize = 16;
        end;
      when '11'
        if HaveFP16Ext() then
          fltsize = 16;
        else
          UNDEFINED;
        end;
      otherwise
        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'        // FCVTSS
        if HaveFCVTSSExt() then UNDEFINED;
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
      otherwise
        UNDEFINED;
    end;
  otherwise
    UNDEFINED;
endcase;

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;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);  // FPRounding_TIEAWAY
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
  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(d = intval<intsize-31:0>, 64);
```

(old)  htmldiff from-  (new)
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 1  | 0  | 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 = \texttt{Uint}(Rd);
integer n = \texttt{Uint}(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

\begin{verbatim}
case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if 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'        // FCVTSS
      if !HaveFP16Ext() then UNDEFINED;
      rounding = FPRounding_TIEAWAY;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI_JS;
      if HaveFP16Ext() then
        fltsize = 16;
      else
        UNDEFINED;
    otherwise
      UNDEFINED;
\end{verbatim}

Assembler Symbols

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

FCVTAU (scalar)
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
   when FPConvOp_CVT_FtoI
      fltval = V[n];
   intval = FPToFixed(fltval, 0, TRUE, FPCR, fltval, 0, unsigned, FPCR, rounding); FPRounding_TIEAWAY);
   when FPConvOp_CVT_ItoF
      intval = X[n];
      fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
   when FPConvOp_MOV_FtoI
      intval = Vpart[n,part];
      fltval = ZeroExtend(intval, 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(d = intval, intval<31:0>, 64);
```

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

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 == 0)

FCVTMS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && type == 0)

FCVTMS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && type == 1)

FCVTMS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && type == 1)

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'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
when '11'
    if HaveFP16Ext() then
        fltsize = 16;
    else
        UNDEFINED;
    rounding = case opcode<2:1>:rmode of
        when '00 xx' // FCVT[INPZ][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;
            if opcode<0> == '1' then
                op = FPConvOp_MOV_ItoF
            else
                op = FPConvOp_MOV_FtoI;
            part = 0;
        when '11 01' // MOV D[1]
            if intsize != 64 || fltsize != 128 then UNDEFINED;
            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(rmode);
        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
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, FALSE, 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(d) = intval<31:0>, 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:50:04

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<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>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></td>
<td></td>
<td></td>
<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>
</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)

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'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
    rounding = 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(rmode);
        otherwise
          UNDEFINED;
  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;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);{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 = V[n];
    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(d = intval[31:0], 64);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:2504

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 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.
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'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
    rounding = 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]CVT
        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' // MOV 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(rmode);
      otherwise
        UNDEFINED;
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

CheckFPAdvSIMEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFIXED(fltval, 0, FALSE, 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
    intval = Vpart[n,part];
    fltval = ZeroExtend(intval, 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 = FPToFIXED2(fltval, FPCR, TRUE);
    X[d] = ZeroExtend[d] = intval<intval<31:0>, 64>;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:50:04

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

\[\text{FCVTNU} \ <Wd>, <Hn>\]

Half-precision to 64-bit (\(sf == 1 \&\& type == 11\))  
(ARMv8.2)

\[\text{FCVTNU} \ <Xd>, <Hn>\]

Single-precision to 32-bit (\(sf == 0 \&\& type == 00\))

\[\text{FCVTNU} \ <Wd>, <Sn>\]

Single-precision to 64-bit (\(sf == 1 \&\& type == 00\))

\[\text{FCVTNU} \ <Xd>, <Sn>\]

Double-precision to 32-bit (\(sf == 0 \&\& type == 01\))

\[\text{FCVTNU} \ <Wd>, <Dn>\]

Double-precision to 64-bit (\(sf == 1 \&\& type == 01\))

\[\text{FCVTNU} \ <Xd>, <Dn>\]
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'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
    rounding = 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(rmode);
    otherwise
      UNDEFINED;
  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;

fltval = case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);{fltval, 0, unsigned, FPCR, rounding};
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        X[d] = fltval;
    when FPConvOp_MOV_FtoI
        intval = V[n];
        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(d = intval4(intval<31:0>, 64));
```
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
case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
      rounding = 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_FtoI;
        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 !HaveFP16Ext() then UNDEFINED;
          rounding = FPRounding_ZERO;
          unsigned = (opcode<0> == '1');
          op = FPConvOp_CVT_FtoI_JS(rmode);
          otherwise
            UNDEFINED;
  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;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, FALSE, 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(d) = intval<intval<31:0>, 64>;

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

(old) htmldiff from- (new)
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.

```
31 30 29 28 27 26 25 24 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 1 0 0 1 0 0 0 0 0 0
    rmode       opcode
```
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'
        UNDEFINED;
        if opcode<2:1>:rmode != '11 01' then UNDEFINED;
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else UNDEFINED;
        rounding = 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_FtoI;
            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(rmode);
            otherwise
                UNDEFINED;
        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;

fltval = case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);{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(d = intval<31:0>, 64);

```
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</th>
<th>0</th>
<th>1</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>0</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></td>
<td></td>
</tr>
</tbody>
</table>

rmode opcode
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 fltsize;

case type of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if 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 case;
      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 case;
    end if;
  end if;
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.

<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;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, fracbits, FALSE, FPCR, FPRounding_ZERO, FPCR, rounding);
  [d] = intval;

  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
  [d] = fltval;
```

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

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.
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 = if opcode<2:1>:rmode != '11 01' then UNDEFINED;

FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  rounding = 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(rmode);
    otherwise
      UNDEFINED;
endcase type;

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

```
CheckFPAdvSIMEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
  X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOVC_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOVC_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(d = intval<31:0>, 64);
```

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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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| 0  | 1  | 1  | 0  | 0  | 1  | scale| Rn | Rd |
---|----|----|----|----|----|----|-----|----|----|----|----|----|----|-----|----|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
```
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;

case type of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
      if 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;
              integer fracbits = 64 - UInt(scale);
        end case;
        if sf == '0' && scale<5> == '0' then UNDEFINED;
        integer fracbits = intsize - UInt(scale);
      end if
    end if
    integer d = UInt(Rd);
    integer n = UInt(Rn);
    integer intsize = if sf == '1' then 64 else 32;
    integer fltsize;
    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
      end case;
      if sf == '0' && scale<5> == '0' then UNDEFINED;
      integer fracbits = 64 - UInt(scale);
    end case
  end if
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;

fltval = case op of
  when FPConvOp_CVT_FtoI => fltval = V[n];
  intval = FPToFixed(fltval, fracbits, TRUE, FPCR, (fltval, fracbits, unsigned, FPCR, rounding); FP rounding);
end when;

V[d] = intval;
V[d] = fltval;
```

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.

<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>1</th>
<th>1</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>
<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)

    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'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
    if '11'
      if HaveFP16Ext() then
        fltsize = 16;
      else
        UNDEFINED;
    else
      UNDEFINED;
  rounding = 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 HaveFP16Ext() then UNDEFINED;
      rounding = FPRounding_ZERO;
      unsigned = (opcode<0> == '1');
      op = FPConvOp_CVT_FtoI_JS(rmode);
      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;

fltval = case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    fltval = intval = V[n];
    intval = FixedToFP(fltval, 0, unsigned, FPCR, rounding);
    X[d] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
  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
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = ZeroExtend((intval<31:0>, 64));
```

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

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

Double-precision to 32-bit
(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>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>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>
</tr>
</tbody>
</table>

sf type rmode opcode

Rn Rd
Double-precision to 32-bit

FJCVTZS <Wd>, <Dn>

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

if 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 HaveFP16Ext() then
        fltsize = 16;
    else
        UNDEFINED;
when '11'
    if HaveFJCVTZSExt() then
        fltsize = 16;
    else
        UNDEFINED;

case opcode2:1:rmode of
when '00' xx'        // FCVT[PMZ][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_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(); then UNDEFINED;
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.
```c
CheckFPAdvSIMDEnabled64();

bits(64) fltval;
bits(32) intval;
bits(fltsize) fltval;
bits(intsize) intval;

fltval = 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-00b08_rc3; Build timestamp: 2018-09-13T13:25Z

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

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 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 CP14C_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  | type | 0  | Rm | 0  | Ra | Rn | Rd |
| o1 | o0 |

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>

```
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 op1_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.
<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 addend, encoded in the "Ra" field.

Operation

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = if opa_neg then operanda = FPNeg(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);
V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | 0 | 1 | 1 | 1 | 0 | type | 1 | Rm | 0 | 1 | 0 | 0 | 1 | 0 | Rd |
|---|---|---|---|---|---|---|---|-----|---|---|---|---|---|---|---|---|

**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;
end case;

case op of
when '00' operation = FMaxMinOp_MAX;
when '01' operation = FMaxMinOp_MIN;
when '10' operation = FMaxMinOp_MAXNUM;
when '11' operation = FMaxMinOp_MINNUM;
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 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];
result = case operation of
    FPMaxMinOp_MAX  result = FPMax(operand1, operand2, FPCR);
    FPMaxMinOp_MIN  result = FPMin(operand1, operand2, FPCR);
    FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
    FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);
V[d] = result;
```

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

```
0 0 0 0 1 1 1 1 0 type 1 | Rd 0 1 1 0 1 0 | Rn | Rd
```

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
      datasize = 16;
    else
      UNDEFINED;
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 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.
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();
b bits(datasize) result;
b bits(datasize) operand1 = V[n];
b bits(datasize) operand2 = V[m];
result = 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;
```

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |

o1 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  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |

o1

**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.

\(<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

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);ReduceOp_FMAXNUM, operand, esize);
```

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

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)

<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>1</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>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>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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<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>0</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>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>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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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”:

<table>
<thead>
<tr>
<th>0</th>
<th>S</th>
</tr>
</thead>
<tbody>
<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);ReduceOp_FMAXNUM, operand, esize);
```

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

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 0 0 0 1 1 1 1 1 0 Rn 0 1 1 1 1 1 1 1 0 Rd
```

**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>

**Single-precision and double-precision**

```assembly
31 30 29 28 27 26 25 24 23 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 0 1 1 1 1 1 1 0 Rn 0 1 1 1 1 1 1 1 0 Rd
```

```assembly
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32;
integer dsize = esize * 2;
integer elements = 2; ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

For the single-precision and double-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></td>
</tr>
<tr>
<td>sz</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>----</td>
<td>-----</td>
</tr>
<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>&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);ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_re3 ; Build timestamp: 2018-09-13T13:25:04

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | Rn  | Rd  |
| o1 |

**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 = 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  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | Rn  | Rd  |
| o1 |

**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 = 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>&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.
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
cHECKFPAvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce[(op, operand, esize);ReduceOp_FMAX, operand, esize];
```

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

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 \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.

31 30 29 28 27 26 25 24 23 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 \textbf{type} 1 0 1 0 1 1 0 \textbf{Rm} \textbf{Rn} \textbf{Rd}

Half-precision (type == 11)
(ARMv8.2)

\begin{verbatim}
FMIN <Hd>, <Hn>, <Hm>
\end{verbatim}

Single-precision (type == 00)

\begin{verbatim}
FMIN <Sd>, <Sn>, <Sm>
\end{verbatim}

Double-precision (type == 01)

\begin{verbatim}
FMIN <Dd>, <Dn>, <Dm>
\end{verbatim}

integer d = \textit{UInt}(Rd);
integer n = \textit{UInt}(Rn);
integer m = \textit{UInt}(Rm);

integer datasize;
case type of
when '00' datasize = 32;
when '01' datasize = 64;
when '10' UNDEFINED;
when '11'
  if \textit{HaveFP16Ext}() then
datasize = 16;
else
  UNDEFINED;
end case;
case op of
when '00' operation = \textit{FPMaxMinOp\_MAX};
when '01' operation = \textit{FPMaxMinOp\_MIN};
when '10' operation = \textit{FPMaxMinOp\_MAXNUM};
when '11' operation = \textit{FPMaxMinOp\_MINNUM};

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.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

FMIN (scalar)
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];

result = 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;
```

(Folded to fit page)
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 \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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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 | 0 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

**Half-precision (type == 11)**

(ARMv8.2)

FMINNM \(<Hd>, <Hn>, <Hm>\)

**Single-precision (type == 00)**

FMINNM \(<Sd>, <Sn>, <Sm>\)

**Double-precision (type == 01)**

FMINNM \(<Dd>, <Dn>, <Dm>\)

\[
\text{integer } d = \text{UInt}(Rd);
\text{integer } n = \text{UInt}(Rn);
\text{integer } m = \text{UInt}(Rm);
\]

\[
\text{integer } datasize;
\text{case } type \text{ of}
\text{when } '00' \text{ datasize } = 32;
\text{when } '01' \text{ datasize } = 64;
\text{when } '10' \text{ UNDEFINED;}
\text{when } '11' \text{ if HaveFP16Ext() then}
\text{ datasize } = 16;
\text{else}
\text{ UNDEFINED; FPMaxMinOp_operation;}
\text{case } op \text{ of}
\text{when } '00' \text{ operation } = \text{FPMaxMinOp_MAX};
\text{when } '01' \text{ operation } = \text{FPMaxMinOp_MIN};
\text{when } '10' \text{ operation } = \text{FPMaxMinOp_MAXNUM};
\text{when } '11' \text{ operation } = \text{FPMaxMinOp_MINNUM};
\]

**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.
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];
result = case operation of
  when FPMINMOp_MAX  result = FPMax(operand1, operand2, FPCR);
  when FPMINMOp_MIN  result = FPMin(operand1, operand2, FPCR);
  when FPMINMOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
  when FPMINMOp_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: 2018-09-13T13:25:04

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

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

<table>
<thead>
<tr>
<th>o1</th>
<th>sz</th>
</tr>
</thead>
</table>

Half-precision

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

if !HaveFP16Ext () then UNDEFINED;

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

integer esize = 16;
integer datasize = 32; 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

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 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   | 1   | 0   | 0   | 0   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | Rn  |   |   |   |   |   |   | Rd |
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

| o1  | sz |

Single-precision and double-precision

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

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

integer esize = 32;
integer datasize = 64; 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

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the half-precision variant: is the destination width specifier, encoded in “sz”:

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.

<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**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);ReduceOp_FMINNUM, operand, esize);
```

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

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

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 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)

```
0 Q 0 0 1 1 1 0 1 0 1 1 0 0 0 0 1 1 0 0 1 0    Rn   Rd
```

**Single-precision and double-precision**

```
0 Q 1 0 1 1 1 0 1 sz 1 1 0 0 0 0 1 1 0 0 1 0    Rn   Rd
```

**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>&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] = ReduceOp_FMINNUM, operand, esize);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:504

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)

<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>1</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>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>
</tr>
</tbody>
</table>

#### Half-precision

FMINP

```c
if !HaveFP16Ext() then UNDEFINED;

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

integer esize = 16;
integer datasize = 32; 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

<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>0</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>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>
</tr>
</tbody>
</table>

#### Single-precision and double-precision

FMINP

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

integer esize = 32;
integer datasize = esize * 2;
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;
```

### Assembler Symbols

<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”:

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>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>&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);ReduceOp_FMIN, operand, esize);
```

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

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

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 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |
| o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Half-precision**

FMINV <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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |
| o1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Single-precision and double-precision**

FMINV <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

**<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>&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.
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);ReduceOp_FMIN, operand, esize);
```

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

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLAL, FMLAL2 (by element)

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 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: FMLAL and FMLAL2

FMLAL
(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  | 1  | 0  | L  | M  | Rm | 0  | 0  | 0  | 0  | H  | 0  | Rn | Rd |

FMLAL2
(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  | 1  | 0  | L  | M  | Rm | 1  | 0  | 0  | 0  | H  | 0  | Rn | Rd |

FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[index]}

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

\[
\text{FMLAL2 } <V_d>, <Ta>, <V_n>, <Tb>, <V_m>. H[<index>] \\
\]

if \(!\text{HaveFP16MulNoRoundingToFP32Ext}()\) then UNDEFINED;
integer \(d = \text{UInt}(R_d)\);
integer \(n = \text{UInt}(R_n)\);
integer \(m = \text{UInt}(\text{0':R_m}); \quad // \text{V_m can only be in bottom 16 registers.}\)
if \(sz == \text{1'}\) then UNDEFINED;
integer index = \text{UInt}(H:L:M);
\[
\]
integer esize = 32;
integer datasize = if \(Q == \text{1'}\) then 128 else 64;
integer elements = datasize \text{ 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.
\(<Ta>\) 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>

\(<V_n>\) 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>

\(<V_m>\) 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

\[\text{CheckFPAdvSIMDEnabled64();}\]
bits(datasize DIV 2) operand1 = \(v_{part}[n, part]\);

\[\text{im,part}\] bits(128) operand2 = \(v[m]\);
bits(datasize) operand3 = \(v[d]\);
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = \(\text{Elem}[\text{operand2, index, esize DIV 2}];\)

for \(e = 0\) to elements-1
\[
\text{element1} = \text{Elem}[\text{operand1, } e, \text{ esize DIV 2}];
\]
if sub_op then \(\text{element1} = \text{FPNeg(element1)};\)
\[
\text{Elem} [\text{result, e, esize}] = \text{FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR)};
\]
\(v[d] = \text{result};\)
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. ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL

(ARMv8.2)

| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | Rm | 1 | 1 | 1 | 0 | 1 | 1 | Rn | Rd |
| S | sz |

FMLAL

FMLAL <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;

FMLAL2

(ARMv8.2)

| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | Rm | 1 | 1 | 0 | 0 | 1 | 1 | Rn | Rd |
| S | sz |

FMLAL2

FMLAL2 <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>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.

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: 2018-09-13T13:25:04

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)

```
| Q | 0 | 1 | 1 | 1 | 1 | 0 | L | M | Rm | 0 | 1 | 0 | H | 0 | Rn | Rd |
```

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:Li: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)

```
| Q | 1 | 0 | 1 | 1 | 1 | 1 | 0 | L | M | Rm | 1 | 1 | 0 | H | 0 | Rn | Rd |
```

FMLSL, FMLSL2 (by element)
FMLSL2

FMLSL2 <Vd>, <Ta>, <Vn>, <Tb>, <Vm>.H[index]

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 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)

31 30 29 28 27 26 25 24 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 | 1 | Rm | 1 | 1 | 1 | 0 | 1 | 1 | Rn | Rd |

S sz

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)

31 30 29 28 27 26 25 24 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 | 1 | 0 | 1 | Rm | 1 | 1 | 0 | 0 | 1 | 1 | Rn | Rd |

S sz

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>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.

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;
```

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)

<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>0</td>
<td>0</td>
<td>0</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Half-precision**

FMOV <Vd>,<T>, #<imm>

if !HaveFP16Ext() then UNDEFINED;

integer rd = UInt(Rd);

integer datasize = if Q == 'l' 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**

<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>op</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>a</td>
<td>b</td>
<td>c</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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;

if cmode:op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;

imm64 = bits(64) imm64;
operation = 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;

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3; Build timestamp: 2018-09-13T13:25Z.
FMOV (vector, immediate)
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.

<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</td>
</tr>
<tr>
<td>opc</td>
</tr>
</tbody>
</table>

Half-precision (type == 11)
(ARMv8.2)

FMOV <Hd>, <Hn>

Single-precision (type == 00)

FMOV <Sd>, <Sn>

Double-precision (type == 01)

FMOV <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
data size = 16;
        else
            UNDEFINED;
        end
end

case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_SQRT;
    when '11' fpop = FPUnaryOp_DST2;
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] = operand; [d] = result;
```

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

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 \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.
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 = 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;
    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 32-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.
Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Is the name of the SIMD&FP source register, encoded in the "Rn" 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(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: 2018-09-13T13:2504)

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

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.

### Half-precision (type == 11) (ARMv8.2)

```
FMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

### Single-precision (type == 00)

```
FMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

### Double-precision (type == 01)

```
FMSUB <Dd>, <Dn>, <Dm>, <Da>
```

```c
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 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.
Operation

```cpp
CheckFPAdvSIMDEnabled64();
bias(datasize) result;
biasdatasize) operanda = V[a];
biasdatasize) operand1 = V[n];
bias(datasize) operand2 = V[m];

operand1 = if olo_neg then operando = FPNeg(operand1);
result = if olo_neg then operando = FPNeg(operand1);
result = FP mulAdd(operand1, operand1, operand2, 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.

<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></td>
<td>(type</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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>
</tr>
</tbody>
</table>

**Half-precision (type == 11)**

(ARMv8.2)

\[
\text{FMUL } <\text{Hd}>, <\text{Hn}>, <\text{Hm}>
\]

**Single-precision (type == 00)**

\[
\text{FMUL } <\text{Sd}>, <\text{Sn}>, <\text{Sm}>
\]

**Double-precision (type == 01)**

\[
\text{FMUL } <\text{Dd}>, <\text{Dn}>, <\text{Dm}>
\]

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

integer datasize;

\[
\text{case type of}
\]

| when '00' datasize = 32; |
| when '01' datasize = 64; |
| when '10' UNDEFINED; |
| when '11' |

\[
\text{if } \text{HaveFP16Ext}() \text{ then} \quad \text{datasize} = 16;
\]

\[
\text{else}\quad \text{UNDEFINED}; \text{ UNDEFINED;}
\]

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

```c
CheckFPAdvSIMDEnabled64();
bias(datasize) result;
bias(datasize) operand1 = V[n];
bias(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);
if negated then result = FPNeg(result);
V[d] = result;
```

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 `CPACR_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>1</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>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>

**Half-precision (type == 11)**
(ARMv8.2)

\[
\text{FNEG } \langle Hd \rangle, \langle Hn \rangle
\]

**Single-precision (type == 00)**

\[
\text{FNEG } \langle Sd \rangle, \langle Sn \rangle
\]

**Double-precision (type == 01)**

\[
\text{FNEG } \langle Dd \rangle, \langle Dn \rangle
\]

integer \( d = \text{UInt}(Rd) \);

integer \( n = \text{UInt}(Rn) \);

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
end

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
CheckFPAdvSIMEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = 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: 2018-09-13T13:25

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 \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  | 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  | 1  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| o1 | o0 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Half-precision (type == 11)
(ARMv8.2)

\texttt{FNMADD} <Hd>, <Hn>, <Hm>, <Ha>

Single-precision (type == 00)

\texttt{FNMADD} <Sd>, <Sn>, <Sm>, <Sa>

Double-precision (type == 01)

\texttt{FNMADD} <Dd>, <Dn>, <Dm>, <Da>

integer d = \texttt{UInt}(Rd);
integer a = \texttt{UInt}(Ra);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if \texttt{HaveFP16Ext}() then
data = 16;
    else
      \texttt{UNDEFINED};
    end

boolean opa_neg = (o1 == '1');
boolean op1_neg = (o0 != o1);

Assembler Symbols

- \texttt{<Dd>} Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Dn>} Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- \texttt{<Dm>} Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- \texttt{<Da>} Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- \texttt{<Hd>} Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Hn>} Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- \texttt{<Hm>} Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- \texttt{<Ha>} Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- \texttt{<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 addend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMEnabled64();
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: 2018-09-13T13:25:04

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

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.

<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>1</td>
<td>1</td>
<td>1</td>
<td>type</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>Ra</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>
<tr>
<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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision (type == 11)
(ARMv8.2)

FNMSUB <Hd>, <Hn>, <Hm>, <Ha>

Single-precision (type == 00)

FNMSUB <Sd>, <Sn>, <Sm>, <Sa>

Double-precision (type == 01)

FNMSUB <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;
    end;
boolean opa_neg = (o1 == '1');
boolean op1_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];

operand = if opa_neg then FPNeg(operand);
result = if op1_neg then FPMulAdd(operand, operand1, operand2, FPCR);
V[d] = result;
```

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

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \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, 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 | 1  | 0  | 0  | 0  | 1  | 0  | Rd |

Half-precision (type == 11)
(ARMv8.2)

\texttt{FNMUL <Hd>, <Hn>, <Hm>}

Single-precision (type == 00)

\texttt{FNMUL <Sd>, <Sn>, <Sm>}

Double-precision (type == 01)

\texttt{FNMUL <Dd>, <Db>, <Dd>}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

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}
  \textbf{UNDEFINED; UNDEFINED;}
\textbf{boolean} negated = (op == '1');
\end{verbatim}

Assembler Symbols

- \texttt{<Dd>} Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Dn>} Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Dm>} Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- \texttt{<Hd>} Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Hn>} Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Hm>} Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- \texttt{<Sd>} Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Sn>} Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<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 = Y[n];
bits(datasize) operand2 = Y[m];

result = FPMul(operand1, operand2, FPCR);
result = if negated then result = FPNeg(result);
V[d] = result;
```
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  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Rn \quad Rd

type \quad op

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> \quad Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dr> \quad Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dr> \quad Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dr> \quad 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 = FPRoundIntN(operand, FPCR, rounding, 32); (operand, FPCR, rounding, intsize);
V[d] = result;
```

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

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

(ajax) htmldiff from- (new)
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, 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  | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | Rn  | Rd  |

Single-precision (type == 00)

FRINT32Z <Sd>, <Sn>

Double-precision (type == 01)

FRINT32Z <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;
endcase

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, op<1>, FPCR, rounding, intsize); FPRounding ZERO, 32);
V[d] = result;
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  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | Rn |
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | Rd |

```
 FRINT64X <Sd>, <Sn>
 Double-precision (type == 01)
 FRINT64X <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;
endcase

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

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundIntN(operand, FPCR, rounding, 64);(operand, FPCR, rounding, intsize);

V[d] = result;
```

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | x  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | Rn | Rd |

type   op

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;
endcase

integer intsize = if op<1> == '0' then 32 else 64;
FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundIntN(operand, FPCR, op<1>, FPRounding, intsize); FPRounding ZERO, 64);
V[d] = result;

Assembler Symbols

<Dd> Is the 64-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.
<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, op<1>, FPRounding, intsize); FPRounding ZERO, 64);
V[d] = result;
(old) htmldiff from- (new)
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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |
| rmode |
```

Half-precision (type == 11)
(ARMv8.2)

FRINTA <Hd>, <Hn>

Single-precision (type == 00)

FRINTA <Sd>, <Sn>

Double-precision (type == 01)

FRINTA <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;
end case;

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.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, FPRounding_TIEAWAY, FALSE);
V[d] = result;
```

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | Rn  | Rd  |

rmode

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;
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 exact = FALSE;
FPRounding rounding;
rounding = case rmode of
when '00x' rounding = FPDecodeRounding(rmode<1:0>);
when '100' rounding = FPRounding_MAXEQU;
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.
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

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, FALSE);(operand, FPCR, rounding, exact);

V[d] = result;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 on the CPACR_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 0 0 1 1 1 1 1 1</th>
<th>type</th>
<th>1 0 0 1 0 1 0 0 0 0</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 0 0 0</td>
<td>Rn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

rmode

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; UNDEFINED;
end case;

boolean exact = FALSE;
FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPDecodeRounding('10');
  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.
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

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, FALSE);(operand, FPCR, rounding, exact);

V[d] = result;
```

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

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

### Half-precision (type == 11)

(ARMv8.2)

```plaintext
FRINTN <Hd>, <Hn>
```

### Single-precision (type == 00)

```plaintext
FRINTN <Sd>, <Sn>
```

### Double-precision (type == 01)

```plaintext
FRINTN <Dd>, <Dn>
```

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

integer datasize;

<table>
<thead>
<tr>
<th>case type of</th>
</tr>
</thead>
<tbody>
<tr>
<td>when '00' datasize = 32;</td>
</tr>
<tr>
<td>when '01' datasize = 64;</td>
</tr>
<tr>
<td>when '10' UNDEFINED;</td>
</tr>
<tr>
<td>when '11'</td>
</tr>
<tr>
<td>if HaveFP16Ext() then</td>
</tr>
<tr>
<td>datasize = 16;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>UNDEFINED;</td>
</tr>
</tbody>
</table>

boolean exact = FALSE;

FPRounding rounding;

<table>
<thead>
<tr>
<th>rounding =case rmode of</th>
</tr>
</thead>
<tbody>
<tr>
<td>when '00x' rounding = FPDecodeRounding('00'); rmode&lt;1:0&gt;1;</td>
</tr>
<tr>
<td>when '100' rounding = FPRounding_TIEAWAY;</td>
</tr>
<tr>
<td>when '101' UNDEFINED;</td>
</tr>
<tr>
<td>when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;</td>
</tr>
<tr>
<td>when '111' rounding = FPRoundingMode(FPCR);</td>
</tr>
</tbody>
</table>
```

### 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.
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

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, FALSE);(operand, FPCR, rounding, exact);
V[d] = result;
```

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

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |

rmode

Half-precision (type == 11) (ARMv8.2)

FRINTP <Hd>, <Hn>

Single-precision (type == 00)

FRINTP <Sd>, <Sn>

Double-precision (type == 01)

FRINTP <Dd>, <Dn>

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

integer datasync;
case type of
  when '00' datasync = 32;
  when '01' datasync = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
dataasync = 16;
    else
      UNDEFINED; UNDEFINED;
else
  UNDEFINED; UNDEFINED;

boolean exact = FALSE;
FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPDecodeRounding('01.'); rmode<1:0>1;
  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, FALSE);
operand, FPCR, rounding, exact);

V[d] = result;
```

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

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.

### Half-precision (type == 11)

(ARMv8.2)

\[
\text{FRINTX } \langle Hd \rangle, \langle Hn \rangle
\]

### Single-precision (type == 00)

\[
\text{FRINTX } \langle Sd \rangle, \langle Sn \rangle
\]

### Double-precision (type == 01)

\[
\text{FRINTX } \langle Dd \rangle, \langle Dn \rangle
\]

```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; UNDEFINED;

Boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
    when '00' 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**

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, TRUE);

V[d] = result;
```

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

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

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 HaveFPl6Ext() then
dataize = 16;
    else
    UNDEFINED; UNDEFINED;
  end if;
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '00' rounding = FPDecodeRounding('11'); 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.
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

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, FALSE);(operand, FPCR, rounding, exact);

V[d] = result;
```

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

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 \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}
0 0 0 1 1 1 0 type 1 0 0 0 0 1 1 1 0 0 0 0 0 | Rn | Rd
\end{verbatim}

Half-precision (type == 11)  
(ARMv8.2)

\begin{verbatim}
FSQRT <Hd>, <Hn>
\end{verbatim}

Single-precision (type == 00)

\begin{verbatim}
FSQRT <Sd>, <Sn>
\end{verbatim}

Double-precision (type == 01)

\begin{verbatim}
FSQRT <Dd>, <Dn>
\end{verbatim}

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

integer datasize;
\quad \text{case type of}
\quad \text{when '00' datasize = 32;}
\quad \text{when '01' datasize = 64;}
\quad \text{when '10' UNDEFINED;}
\quad \text{when '11'
\quad \quad \quad \text{if HaveFP16Ext() then}
\quad \quad \quad \quad \text{datasize = 16;}
\quad \quad \quad \text{else}
\quad \quad \quad \quad \quad \text{UNDEFINED;}
\quad \quad \quad \quad \quad \text{FPUnaryOp fpop}
\quad \quad \quad \text{case opc of}
\quad \quad \quad \quad \text{when '00' fpop = FPUnaryOp\_MOV;}
\quad \quad \quad \quad \text{when '01' fpop = FPUnaryOp\_ABS;}
\quad \quad \quad \quad \text{when '10' fpop = FPUnaryOp\_NEG;}
\quad \quad \quad \quad \text{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 CheckFPAdvSIMEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = case fpop of
  when FPUnaryOp_MOV result = operand;
  when FPUnaryOp_ADD result = FPAbs(operand);
  when FPUnaryOp_SUB 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: 2018-09-13T13:25:04

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

### 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>
```

```plaintext
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;
  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];
result = 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: 2018-09-13T13:25

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.

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' see "XPACLRI";
  when '0001 xxx' 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' see "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP"
  when '0100 xx0' op = SystemHintOp_BTI;
  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_YIELDHint_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_SEVSendEvent();
  when SystemHintOp_SEVLSendEventLocal();
  when SystemHintOp_ESBSynchronizeErrors();
    AArch64.ESBOperation();
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
      AArch64.vESBOperation();
      TakeUnmaskedErrorInterrupts();
  when SystemHintOp_PSBProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDBCConsumptionOfSpeculativeDataBarrier();
  otherwise    // do nothing
    when SystemHintOp_BT
      BTypeNext = '00';
  otherwise // do nothing

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

Halt instruction. A 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 |
|------------------------------------------|----------|
| imm16                                    |imm 16    |
```

System

```
HLT #<imm>

if EDSCR.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);
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8_rc3 ; Build timestamp: 2018-09-13T13:25Z4

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

System

\[ \text{HVC} \#<\text{imm}> \]

// Empty, 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[(imm16)+[imm]];
**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 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(128) result;
bits(datasize) result;
result = V[d];
Elem[result, index, 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.
IRG

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)

<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>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>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>
</tr>
</tbody>
</table>

Integer

IRG <Xd|SP>, <Xn|SP>{, <Xm>}

| integer d = UInt(Xd); |
| integer n = UInt(Xn); |
| integer m = UInt(Xm); |

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

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
        RGSR_EL1 = bits(32) UNKNOWN;
        rtag = _ChooseRandomNonExcludedTag(exclude);
    else
        bits(4) start = RGSR_EL1.TAG;
        bits(4) offset = RandomTag();
        rtag = ChooseNonExcludedTag(start, offset, exclude);
    else
        rtag = '0000';

bits(64) result = AddressWithAllocationTag(operand, rtag);
if d == 31 then
    SP[] = result;
else
    X[d] = result;

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Instruction Synchronization Barrier (ISB)

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 1 1 0 0 1 1 CRm 1 1 0 1 1 1 1 opc
```

System

ISB {<option>|#<imm>}

```c
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

op = case opc of
  when "00" op = MemBarrierOp_DSB;
  when "01" op = MemBarrierOp_DMB;
  when "10" op = MemBarrierOp_ISB;
  case CRm<3:2> of
    when "00" domain = 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_SBB;
          elsif HaveSBExt() && FALSE then
            if CRm<3:2> == "00" && opc == "11" then
              op = MemBarrierOp_SB;
            else
              types = MBReqTypes_All;
              domain = MBReqDomain_FullSystem;
        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_DSBDataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier();
  when MemBarrierOp_SSBSpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBSpeculativeSynchronizationBarrierToPA();
  when MemBarrierOp_SBSpeculationBarrier();
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  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | x  | size | Rn |    |    |    |    |    |    |    |    |    |

L 0

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  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  |    |    |    |    |    |    | size | Rn |    |    |    |    |    |    |    |    |    |

L 0

opcode
One register, immediate offset (Rm == 11111 && opcode == 0111)

LD1 { <Vt>.<T> }, [Xn|SP], <imm>

One register, register offset (Rm != 11111 && opcode == 0111)

LD1 { <Vt>.<T> }, [Xn|SP], <Xm>

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

LD1 { <Vt>.<T>, <Vt2>.<T> }, [Xn|SP], <imm>

Two registers, register offset (Rm != 11111 && opcode == 1010)

LD1 { <Vt>.<T>, <Vt2>.<T> }, [Xn|SP], <Xm>

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP], <imm>

Three registers, register offset (Rm != 11111 && opcode == 0110)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP], <Xm>

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [Xn|SP], <imm>

Four registers, register offset (Rm != 11111 && 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”: 

<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 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>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</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>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</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>#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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp STORE
                Mem[address+offs, ebytes, 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 CPACR_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  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | x  | x  | 0  | S  | size | Rn | Rt |

L R opcode

8-bit (opcode == 000)

LD1 { <Vt>.B }[<index>], [<Xn|SP>]  

16-bit (opcode == 010 && size == x0)

LD1 { <Vt>.H }[<index>], [<Xn|SP>]  

32-bit (opcode == 100 && size == 00)

LD1 { <Vt>.S }[<index>], [<Xn|SP>]  

64-bit (opcode == 100 && S == 0 && size == 01)

LD1 { <Vt>.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  | 1  | 0  | Rm | x  | x  | 0  | S  | size | Rn | Rt |

L R opcode
8-bit, immediate offset (Rm == 11111 && opcode == 000)

```assembly
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```assembly
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```assembly
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```assembly
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```assembly
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```assembly
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```assembly
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```assembly
LD1 { <Vt>.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.
- `<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, 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, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, 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.


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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 1  | 1  | 0  | 0  | size | Rn | Rt |

L R opcode S

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;

### Post-index

Immediate offset (Rm == 1111)

LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 1111)

LD1R { <Vt>.<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>

**<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>#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 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, 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, address + offs, ebytes, AccType_VEC];
         V[t] = rval;
      else // memop == MemOp_STORE
         // extract from one lane of 128-bit register
         Mem[address+offs, ebytes, 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**

| 31 30 29 28 27 26 25 24 23 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 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | size | Rn | Rt |
| L | opcode |

**Post-index**

LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>] 

integer t = UInt(Rt); 
integer n = UInt(Rn); 
integer m = integer UNKNOWN; 
boolean wback = FALSE;

**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**

```
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 = 3; selem = 1; // LD/ST1 (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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | x  | x  | 0  | S  | size | Rn | Rt |

L R opcode

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>]

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  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | Rm | x  | x  | 0  | S  | size | Rn | Rt |

L R opcode
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.
Shared Decode

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];
off = zeros();
if replicate then
// load and replicate to all elements
for s = 0 to selem-1
  element = Mem[address+offs, ebytes, 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, address + offs, ebytes, AccType_VEC];
    V[t] = rval;
  else // memop == MemOp_STORE
    // extract from one lane of 128-bit register
    Mem[address+offs, ebytes, 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.
LD2R

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 CPACR_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  1  1  0  0  0  0  0  1  1  0  0  size |
|                                | Rn             | Rt             |
| L                               | R              | opcode        | S               |

No offset

```java
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```java
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  1  1  1  1  1  1  Rm  1  1  0  0  size |
|                                | Rn             | Rt             |
| L                               | R              | opcode        | S               |

Immediate offset (Rm == 11111)

```java
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```java
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

```java
<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
```

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

| 31 30 29 28 27 26 25 24 23 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  1  1  1  Rm  1  1  0  0  size |
|                                | Rn             | Rt             |
| L                               | R              | opcode        | S               |

```java
size | Q | <T>
---|---|---
00  | 0 | 8B
00  | 1 | 16B
01  | 0 | 4H
01  | 1 | 8H
10  | 0 | 2S
10  | 1 | 4S
11  | 0 | 1D
11  | 1 | 2D
```

```java
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 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>#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, 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, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOpSTORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, 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   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |

<table>
<thead>
<tr>
<th>L</th>
<th></th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LD3**

\[ \text{LD3} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [Xn|SP] \]

```c
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   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |

<table>
<thead>
<tr>
<th>L</th>
<th></th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Immediate offset (Rm == 11111)**

\[ \text{LD3} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [Xn|SP], <imm> \]

**Register offset (Rm != 11111)**

\[ \text{LD3} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [Xn|SP], <Xm> \]

```c
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.

<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>#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, address + offs, ebytes, AccType_VEC];
        V[tt] = rval;
      else // memop == MemOp_STORE
        Mem[address+offs, ebytes, 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: 2018-09-13T13:25

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

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>]
```

Post-index

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```
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>

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.
&lt;index&gt; 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, 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, address + offs, ebytes, AccType_VEC];
      V[t] = rval;
    else
      // memop == MemOp_STORE
      // extract from one lane of 128-bit register
      Mem[address+offs, ebytes, 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>0</td>
<td>Q</td>
<td>0</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>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>
</tbody>
</table>

L R opcode S

No offset

LD3R { <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

<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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>Rm</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>
</tr>
</tbody>
</table>

L R opcode S

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.

<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":

<table>
<thead>
<tr>
<th>size</th>
<th>imm</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>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding ZXR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
type 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]
    end if
    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;
```

LD3R
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, 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, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, 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, 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

```
0 Q 0 0 1 1 0 0 0 0 0 0 0 0 0 0 size    Rn    Rt
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

```
0 Q 0 0 1 1 0 0 0 0 0 0 0 0 0 0 size    Rn    Rt
```

```
```

### Immediate offset (Rm == 1111)

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 1111)

```
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>
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>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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
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

```
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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, 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 CPACR_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  | 1  | 0  | 1  | 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)


32-bit (opcode == 101 && size == 00)


64-bit (opcode == 101 && S == 0 && size == 01)


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  | 1  | 1  | 1  | Rm | x  | x  | 1  | S  | size | Rn | Rt |

L R opcode
**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)**


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, 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, address + offs, ebytes, AccType_VEC];
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, 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.
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 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

L R  opcode  S

Rn  Rt

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 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | size | Rn | Rt |

L R  opcode  S

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.
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

```python
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, 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, address+offs, ebytes, AccType_VEC];
      V[t] = rval;
    else // memop == MemOp_STORE
      // extract from one lane of 128-bit register
      Mem[address+offs, ebytes, 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)

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 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  | 0   | 0   | 0   | 0   | 0   | Rn  |    |    |    |    |    |    |    |    |    | Rt  |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
</table>

LDADD, LDADDA, LDADDAL, LDADDL
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.
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 == '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
    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];

result = data + value; case op of

    MemAtomicOp_ADD result = data + value;
    MemAtomicOp_BIC result = data AND NOT(value);
    MemAtomicOp_EOR result = data EOR value;
    MemAtomicOp_ORR result = data OR value;
    MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    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.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
**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, LDADDB 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)**

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Ws</th>
<th>Wt</th>
<th>Xn</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**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 <<

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

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.

<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>STADDDB, STADDB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
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, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data + value; case op of
    MemAtomicOp_ADD result = data + value;
    when MemAtomicOp_BIC result = data AND NOT(value);
    when MemAtomicOp_BIC result = data XOR value;
    when MemAtomicOp_ORS result = data OR value;
    when MemAtomicOp_SMX result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMM result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMX result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMM result = if UInt(data) > UInt(value) then value else data;
Mem[address, 1, stacctype] = result;
[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, 32);(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: 2018-09-13T13:25
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>Ws</th>
<th>Wt</th>
<th>Xn</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### 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>]

```plaintext
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); // 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.

<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>STADDH, STADDLH</td>
<td>A == '0' &amp; &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
  CheckSPLAlignment();
  address = SP[];
else
  address = X[n];

// All observers in the shareability domain observe the following load and store atomically.
data = Mem[address, 2, ldacctype];
[address, datasize DIV 8, ldacctype];
result = data + value; case op of
  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, 2, stacctype] = result;
[address, datasize DIV 8, stacctype] = result;

if t != 31 then
  X[t] = ZeroExtend(data, 32);(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, 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.

### 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  | 1  | 0  | 0  | 1  | 0  | 1  | (1)| (1)| (1)| (1)| 1  | 1  | 0  | 0  | 0  | Rn |  |  |  |  |  |  |  |  |  |  |  |
| size|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

#### 32-bit (size == 10)

LDAPR <Wt>, [<Xn|SP> {,#0}]

#### 64-bit (size == 11)

LDAPR <Xt>, [<Xn|SP> {,#0}]

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << Integer(UInt(size));
integer regsize = if elsize == 64 then 64 else 32; (Rs); // ignored by all loads and store-release AccType
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.
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
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, {address, dbytes, acctype}; AccType ORDERED];
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, 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*.

### 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**size**  
Rs

**Rn**  
Rt

### LDAPRB (<Wt>), [<Xn|SP> {,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release

AccType accctype = 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

```plaintext
bits(64) address;
bite(8) data;
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, 1,[address, dbytes, accctype], AccType_ORDERED];
X[t] = ZeroExtend(data, 32);[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, 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*.

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>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>(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>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>Rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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

LDAPRH <Wt>, [<Xn|SP> {,#0}]

```plaintext
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

```plaintext
bits(64) address;
bits(16) data;
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, 2, [address, dbytes, acctype] = AccType_ORDERED];
X[t] = ZeroExtend(data, 32);idata, 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 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*.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 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   | 1   | 0   | imm9 | 0   | 0   | Rn  | Rt  |

32-bit (size == 10)

LDAPUR <Wt>, [<Xn|SP>\{, #<simm>\}]

64-bit (size == 11)

LDAPUR <Xt>, [<Xn|SP>\{, #<simm>\}]

```java
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bzero(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);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
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 rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_NOP EndOfInstruction();
    end

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = MemOp_LOADConstrainUnpredictable();
        SetNotTagCheckedInstruction(is_load_store & n == 31);

    if n == 31 then
        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 rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_NOP EndOfInstruction();
    end

    if n == 31 then
        if memop == MemOp_PREFETCH then CheckSPAlignment();
        else
            address = SP[];
            address = address + offset;
            if !postindex then
                address = address + offset;
            end
            data = case memop of
                when MemOp_STORE
                    if rt_unknown then
                        data = bits(datasize) UNKNOWN;
                    else
                        data = X[t];
                    end
                    Mem(address, datasize DIV 8, address, datasize DIV 8, acctype) = data;
                when AccType_ORDERED MemOp_LOAD
                    data = Mem(address, datasize DIV 8, address, datasize DIV 8, acctype);
                    if signed then
                        X[t] = SignExtend(data, regsize);
                    else
                        X[t] = ZeroExtend(data, regsize);
                    end
            end
            when MemOp_PREFETCH
                Prefetch(address, t<4:0>);
                if wback then
                    address = bits(64) UNKNOWN;
                    elsif postindex then
                        address = address + offset;
                    if n == 31 then
                        SP[] = address;
else
(x(data, regsize), (h) = 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.
LDAPURB

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.

<imm> 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 acc-type = 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;
  else
    if opc<0> == '1' then UNDEFINED;
    else
      // sign-extending load
      memop = MemOp_LOAD;
  end
  if size == '10' || opc<0> == '1' then UNDEFINED;
  else
    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} IN (1,
  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();
  else
    c = MemOp_LOAD ConstrainUnpredictable;
    SetNotTagCheckedInstruction(is_load_store & n == 31);
  end

  if n == 31 then
    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();
  else
    if memop == MemOp_PREFETCH then CheckSPAignment();
    else
      address = X[n];
    end
    address = address + offset;
    if !postindex then
      address = address + offset;
    end
    data = case memop of
      when MemOp_STORE
        if rt_unknown then
          data = bits(datasize) UNKNOWN;
        else
          data = X[t];
        Mem[address, 1, address, datasize DIV 8, acctype] = data;
      when AccType_ORDERED MemOp_LOAD]
data = Mem[address, datasize DIV 8, acctype];
      if signed then
        X[t] = SignExtend(data, regsize);
      else
        X[t] = ZeroExtend(data, regsize);
      Mem[address, 1, address, datasize DIV 8, acctype] = data;
    end
    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;
      end
    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.
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.

Unscaled offset

LDAPURH <Wt>, [<Xn|SP>{, #<simm>}]

```plaintext
bits(64) offset = 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);  
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} IN {MemOp_LOAD, MemOp_STORE};
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 = MemOp_LOADConstrainUnpredictable();
    SetNotTagCheckedInstruction(is_load_store && n == 31);

bits(64) address;
bits(16) data;

if n == 31 then
    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();
else
    address = SP[];

address = address + offset;
if ! postindex then
    address = address + offset;
data = case memop of
        when MemOp_STORE
            if rt unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
        Mem[address, 2];
        case memop of
            when AccType_ORDERED(MemOp_LOAD)
                data = Mem[address, datasize DIV 8, acctype] = data;
            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(data, 32); [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.
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*.

32-bit (opc == 11)

\[ \text{LDAPURSB } <Wt>, [<Xn|SP>{, #<simm}>} \]

64-bit (opc == 10)

\[ \text{LDAPURSB } <Xt>, [<Xn|SP>{, #<simm}>} \]

\[
\begin{array}{cccccccccccc}
\text{bits(64)} & \text{offset} & \text{wback} & \text{false} & \text{postindex} & \text{false} & \text{scale} & \text{Uint(size)} & \text{offset} & \text{SignExtend}(\text{imm9}, 64)\\
\end{array}
\]

**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.
- \( <\text{simm}> \) 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};
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  \texttt{MemOp} = \texttt{if} opc<0> == '1' then \texttt{MemOp\_LOAD} \texttt{else} \texttt{MemOp\_STORE};
  \texttt{regsize} = \texttt{if} size == '11' then 64 \texttt{else} 32;
  \texttt{signed} = FALSE;
else
  \texttt{if} size == '11' then
    \texttt{memop} = \texttt{MemOp\_PREFETCH};
    \texttt{regsize} = 32;
    \texttt{signed} = FALSE;
  \texttt{else}
    \texttt{if} size == '11' then
      \texttt{memop} = \texttt{if} opc<0> == '1' then \texttt{UNDEFINED};
      \texttt{regsize} = \texttt{if} opc<0> == '1' then 32 \texttt{else} 64;
      \texttt{signed} = \texttt{TRUE};
  \texttt{else}
    \texttt{if} size == '10' \texttt{&&} opc<0> == '1' then \texttt{UNDEFINED};
    \texttt{regsize} = \texttt{if} opc<0> == '1' then 32 \texttt{else} 64;
    \texttt{signed} = \texttt{TRUE};

integer datasize = 8 \ll scale;
if \texttt{HaveMTEExt}() then
    boolean \texttt{is\_load\_store} = \texttt{memop} IN \{\texttt{MemOp\_STORE}, \texttt{MemOp\_LOAD}\};
    SetNotTagCheckedInstruction(\texttt{is\_load\_store} \&\& \texttt{n} == 31);
    SetNotTagCheckedInstruction(\texttt{is\_load\_store} \&\& \texttt{t} == 31 \&\& \texttt{wback});

    bits(64) \texttt{address};
    Bits(8) \texttt{data};
    Bits(datasize) \texttt{data};

    if \texttt{n} == 31 then
        if \texttt{memop} != \texttt{MemOp\_LOAD} \&\& \texttt{wback} \&\& \texttt{n} == \texttt{t} \&\& \texttt{n} == 31 then
            \texttt{c} = \texttt{ConstrainUnpredictable\_Unpredictable\_WBOVERLAPLD};
            assert \texttt{c} IN \{\texttt{Constraint\_WBSUPPRESS}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDER}, \texttt{Constraint\_NOP}\};
            case \texttt{c} of
                when \texttt{Constraint\_WBSUPPRESS} \texttt{wback} = \texttt{FALSE}; // writeback is suppressed
                when \texttt{Constraint\_UNKNOWN} \texttt{wb\_unknown} = \texttt{TRUE}; // writeback is UNKNOWN
                when \texttt{Constraint\_UNDER} \texttt{UNDEFINED};
                when \texttt{Constraint\_NOP} \texttt{EndOfInstruction}();
            endcase;
        end

        if \texttt{memop} == \texttt{MemOp\_LOAD} \&\& \texttt{wback} \&\& \texttt{n} == \texttt{t} \&\& \texttt{n} != 31 then
            \texttt{c} = \texttt{ConstrainUnpredictable\_Unpredictable\_WBOVERLAPLD};
            assert \texttt{c} IN \{\texttt{Constraint\_NONE}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDER}, \texttt{Constraint\_NOP}\};
            case \texttt{c} of
                when \texttt{Constraint\_NONE} \texttt{rt\_unknown} = \texttt{FALSE}; // value stored is original value
                when \texttt{Constraint\_UNKNOWN} \texttt{rt\_unknown} = \texttt{TRUE}; // value stored is UNKNOWN
                when \texttt{Constraint\_UNDER} \texttt{UNDEFINED};
                when \texttt{Constraint\_NOP} \texttt{EndOfInstruction}();
            endcase;
        end

        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}];
        end

        \texttt{address} = \texttt{address} + \texttt{offset};
        if \texttt{! postindex} then
            \texttt{address} = \texttt{address} + \texttt{offset};
        end

        case \texttt{memop} of
            when \texttt{MemOp\_STORE}
                \texttt{data} = \texttt{If rt\_unknown then}
                    \texttt{data} = \texttt{bits(datasize) UNKNOWN};
                else
                    \texttt{data} = \texttt{X}[\texttt{t}];
                    \texttt{Mem} (\texttt{address}, 1, \texttt{address}, datasize DIV 8, \texttt{acctype}) = \texttt{data};
                    \texttt{when AccType\_ORDERED} = \texttt{data};
            endcase;

            when \texttt{MemOp\_LOAD}
                \texttt{data} = \texttt{Mem} (\texttt{address}, 1, \texttt{address}, datasize DIV 8, \texttt{acctype});
                if \texttt{signed} then \texttt{AccType\_ORDERED};
                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});
                endcase;
            endcase;

            when \texttt{MemOp\_PREFETCH}
                \texttt{Prefetch} (\texttt{address}, \texttt{t<4:0>});
            endcase;

            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};
                end
            end
        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.
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.

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>}]

Bits(64) offset = 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);
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;
    regsize = 32;
    signed = FALSE;
else
  // sign-extending load
  memop = if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;
if size == '10' && opc<0> == '1' then UNDEFINED;
else
  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);
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;
if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    data = if rt_unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
    Mem[address, 2, address, datasize DIV 8, acctype] = data;
    when AccType_ORDERED] = data;
  when MemOp_LOAD
    data = Mem[address, 2, address, datasize DIV 8, acctype];
    if signed then AccType_ORDERED];
    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[address, t<4:0>];[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.
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.

Unscaled offset

LDAPURSW <Xt>, [<Xn|SP>{, #<simm>}]}

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 = if opc<0> == '1' then MemOp_PREFETCH else MemOp_STORE;
boolean signed = FALSE;
integer regsize = 64;
integer datasize = 8 << scale;
```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD IN {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 = 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 UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE & wback & n == t & n != 31 then
        c = MemOp_LOAD ConstrUnpredictable;
    SetNotTagCheckedInstruction(is_load_store & n == 31);

    bits(64) address;
    bits(32) data;

    if n == 31 then
        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();
        else
            address = X[n];

        address = address + offset;
        if !postindex then
            address = address + offset;

        data = case memop of
            when MemOp_STORE
                if rt_unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = X[t];
            Mem[address, 4, address, datasize DIV 8, acctype] = data;
            when AccType_ORDERED(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_PREFETCHPrefetch(address, t<4:0>);

        if wback then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
else
 else

 (data, 64); \[n\] = 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

**32-bit (size == 10)**

LDAR \(<Wt>, \langle<Xn|SP\\rangle\{,#0\})\

**64-bit (size == 11)**

LDAR \(<Xt>, \langle<Xn|SP\\rangle\{,#0\})\

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << Integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32; (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 << Integer t2 = 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

```c
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
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 = case memop of
    when MemOp_STORE
        data = X[t];
    Mem(address, dbytes, address, dbytes, acctype) = data;
    when
        data = MemAccType_ORDERED
            X[t] = ZeroExtend(data, regsize);
```
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.

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(8) data;
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 = case memop of
    when MemOp_STORE
        data = X[t];
    MemOp_LOAD, dbytes, acctype] = data;
    when
        data = MemAccType_ORDEREDMemOp_LOAD][address, dbytes, acctype];
    X[t] = ZeroExtend(data, 32); 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.
LDARH

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.

### Assembler Symbols

<\(W_t\)> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<\(X_n\)|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(16) data;
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 = case memop of
    MemOp_STORE data = X[t];
    Mem[address, 2], address, dbytes, acctype] = data;
    when
    MemAccType_ORDEREDMemOp_LOAD], address, dbytes, acctype],
    X[t] = ZeroExtend(data, 32); [data, resize);
```

### 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}]

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz); // ignored by load/store single-register
integer datasize = elsize * 2; // 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 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 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 "Rt" 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 LOAD, MemOp STORE};
    SetNotTagCheckedInstruction(is_load_store && n == 31);
endif

if t == t2 then if memop ==
    MemOp LOAD 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 UNDEFINE;
        when Constraint_NOP EndOfInstruction();
endif
endif

if n == 31 then 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 UNDEFINE;
            when Constraint_NOP EndOfInstruction();
        endif
    endif
endif

if n == 31 then
    if memop ==
        MemOp_STORE then
            if t == t2 then
                Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
                assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
            case c of
                when Constraint_UNKNOWN rt_unknown = TRUE;    // address is UNKNOWN
                when Constraint_NONE rt_unknown = FALSE;    // address is original base
                when Constraint_UNDEF UNDEFINE;
                when Constraint_NOP EndOfInstruction();
            endif
        endif
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rt_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
endif

// 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.
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];
        endif
        bit status = '1';
    endif
    // 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;
    endif
endif
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 rt_unknown then
  // ConstrainedUNPREDICTABLE case
  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, {address, dbytes, acctype};
        if AccType_ORDEREDATOMIC]f
        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), AccType_ORDEREDATOMIC);
        else
          data = AccType_ORDEREDATOMIC Mem[address, 8, {address, 8, acctype};
            X[t] = Mem[address, 8, address + 8, 8, acctype];
            X[t2] = Mem[address+8, 8, AccType_ORDEREDATOMIC];
            X[t2] = MemZeroExtend(address+8, 8, AccType_ORDEREDATOMIC];
        end
      end
    end
  end
end
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}]

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(elsize) data;
constant integer dbytes = elsize DIV 8;
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_LOAD IN {MemOp_STORE, MemOp LOAD IN (;
    SetNotTagCheckedInstruction(is_load_store ft n == 31);
  if memop == MemOp LOAD || pair && t == t2 then
    assert c IN {Constraint UNKNOWN, Constraint UNDER, Constraint NOP};
    case c of
      when Constraint UNKNOWN rt unknown = TRUE; // result is UNKNOWN
      when Constraint UNDER UNDEFINED;
      when Constraint NOP EndOfInstruction();
    if memop == MemOp STORE then
      if s == t || (pair && s == t2) then MemOp LOAD [Constraint];
      SetNotTagCheckedInstruction(is_load_store 64 n == 31);
    if n == 31 then
      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
        assert c IN {Constraint UNKNOWN, Constraint NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
          when Constraint UNKNOWN rt unknown = TRUE; // address is UNKNOWN
          when Constraint NONE rt 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];
      // 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. case memop of
      when MemOp STORE
        if rt unknown then
          data = bits(datasize) UNKNOWN;
        else if 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
        LDAXR
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);
```

// 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);
```

```
data = 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, address, dbytes, acctype];
    if () 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 = MemAccType_ORDEREDATOMICBigEndian[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.
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.

No offset

LDAXRB <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(8) data;
bits(datasize) data;

constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD IN {MemOp_LOAD, MemOp_STORE}};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp_LOAD if pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_NONE      rt_unknown = FALSE;   // store original value
        when Constraint_UNDEF     UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
        if n == 31 then
            Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
            assert c IN {Constraint_UNKNOWN, 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];

        // 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. Case memop of
        // when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else 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...
Mem[address, dbytes, acctype] = data;
status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);

when MemOpt_LOAD

// Tell the Exclusive monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive 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, 1);

(address, dbytes);

data = 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, 1, (address, dbytes, acctype)];
        if () 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 = MemAccType_ORDEREDATOMIC[BigEndian];(address, dbytes, acctype,)
            X[t] = ZeroExtend(data, 32);(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 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_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 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(16) data;
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 then
    if pair & t = t2 then
      Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
      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;

    if n == 31 then
      Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
      assert c IN { Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP };
      case c of
        when Constraint_UNKNOWN rn Unknown = TRUE; // address is UNKNOWN
        when Constraint_UNDEF          UNDEFINED;
        when Constraint_NOP           EndOfInstruction();
      end;

      if n == 31 then
        CheckSPAlignment();
        address = SP[];
      elseif rn unknown then
        address = bits(64) UNKNOWN;
      else
        address = X[n];

      if rt Unknown then
        bits(dbytes) data = UNKNOWN;
      elseif pair then
        bits(dbytes \div 2) all = X[t];
        data = if BigEndian() then all : all else all : all;
      else
        data = X[t];

      bit status = '1';
  end;
end;

// 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.
if memop == MemOp_STORE then
  if rt Unknown then
    bits(dbytes) data = UNKNOWN;
  elseif pair then
    bits(dbytes \div 2) all = X[t];
    data = if BigEndian() then all : all else all : all;
  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 = AArch64.ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);

// 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, 2);

(data, dbytes);

data = 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, 2, [address, dbytes, acctype];
    if () then
      X[t] = data<datasize-1:elsize>
    else
      X[t] = data<elsize-1:0>
    end
  else
    // 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));
    else
      data = Mem[address;[address, dbytes, acctype];
      X[t] = ZeroExtend(data, 32);[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.
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)

| 31 30 29 28 27 26 25 24 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 x 1 1 1 0 0 0 A R 1 Rs 0 0 0 1 0 0 | Rn | Rt |
| size | opc |
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 == '1111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
definitions:
    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];
result = data AND NOT(value);
```  

```plaintext
    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.
LDCLRB, 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.
- LDCLRB 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 STCLRB, STCLRLB.

### Integer

(ARMv8.1)

<table>
<thead>
<tr>
<th>Size</th>
<th>Opc</th>
<th>Rs</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### LDCLRAB (A == 1 && R == 0)

LDCLRAB <Ws>, <Wt>, [<Xn|SP]>

### LDCLRALB (A == 1 && R == 1)

LDCLRALB <Ws>, <Wt>, [<Xn|SP]>

### LDCLRB (A == 0 && R == 0)

LDCLRB <Ws>, <Wt>, [<Xn|SP]>

### LDCLRLB (A == 0 && R == 1)

LDCLRLB <Ws>, <Wt>, [<Xn|SP]>

```c
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.

<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>STCLRB, STCLRLB</td>
<td>$A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```
bits(64) address;
bite(8) value;
bite(8) data;
bite(8) result;
bite(datasize) value;
bite(datasize) data;
bite(datasize) result;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];
address, datasize DIV 8, ldacctype];

result = data AND NOT(value);

Mem[address, 1, stacctype] = result;
address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, 32);(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, LDCLRAH, 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, LDCLRAH 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, STCLRLH.

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>A</td>
<td>R</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td>Rs</td>
<td></td>
</tr>
</tbody>
</table>

LDCLRH (A == 1 & R == 0)

LDCLRH <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]>

```glsl
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;
```

LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH
Page 415
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>STCLRH, STCLRLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bites(16) value;
bites(16) data;
bites(16) result;
bites(datasize) value;
bites(datasize) data;
bites(datasize) result;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);
value = X[s];
if n == 31 then
    CheckSPAinement();
    address = SP[];
else
    address = X[n];
// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, ldacctype];
address, datasize DIV 8, ldacctype];
result = data AND NOT(value); case op of
| MemAtomicOp ADD | result = data + value;                  |
| MemAtomicOp BIC | result = data AND NOT(value);           |
| MemAtomicOp POP  | result = data XOR value;                |
| MemAtomicOp ORR | result = data OR value;                 |
| MemAtomicOp SMAX | result = if SInt(data) > SInt(value) then data else value;
| MemAtomicOp SMIN | result = if SInt(data) > SInt(value) then value else data;
| MemAtomicOp UMAX | result = if UInt(data) > UInt(value) then data else value;
| MemAtomicOp UMN | result = if UInt(data) > UInt(value) then value else data;
Mem[address, 2, stacctype] = result;
(address, datasize DIV 8, stacctype] = result;
if t != 31 then
    X[t] = ZeroExtend(data, 32);(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.
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)

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  |   | Rn |   |   |   |   |   |   |   |   |   |   |   |
| size |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opc |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
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.
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;&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];
result = data EOR value; case op of
    MemAtomicOp_ADD result = data + value;
    MemAtomicOp_BIC result = data AND NOT(value);
    MemAtomicOp_EOR result = data EOR value;
    MemAtomicOp_ORR result = data OR value;
    MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
Mem[address, datasize DIV 8, storctype] = 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  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 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);

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.

<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>STEORB, STEORLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bids(8) value;
bids(8) data;
bids(8) result;
bids(datasize) value;
bids(datasize) data;
bids(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, 1, ldacctype];
address, datasize DIV 8, ldacctype];

result = data EOR value; case op of

MemAtomicOp_ADD  result = data + value;
  when MemAtomicOp_ADDC result = data AND NOT(value);

MemAtomicOp_and  result = data AND value;

MemAtomicOp_ANDX result = data AND NOT(value);

MemAtomicOp_EOR  result = data XOR value;

MemAtomicOp_LDSB result = if SInt(data) > SInt(value) then data else value;

MemAtomicOp_MAX  result = if UInt(data) > UInt(value) then data else value;

MemAtomicOp_MIN  result = if UInt(data) > UInt(value) then value else data;

MemAtomicOp_MAXU result = if UInt(data) > UInt(value) then value else data;

MemAtomicOp_MINU result = if UInt(data) > UInt(value) then data else value;

Mem[address, 1, stacctype] = result;
(address, datasize DIV 8, stacctype] = result;

if t != 31 then
  X[t] = ZeroExtend(data, 32);4(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  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | Rn | 0  | 0  | 1  | 0  | 0  | Rt |

**Size**

LDEORAH (A == 1 && R == 0)

LDEORAH <Ws>, <Wt>, [<Xn|SP]>

LDEORALH (A == 1 && R == 1)

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]>

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.

<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>STEORH, STEORLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```c
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
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, 2, ldacctype];
[address, datasize DIV 8, ldacctype];
result = data EOR value; 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_MAX result = if SInt(data) > SInt(value) then data else value;
        when MemAtomicOp_MIN 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, 2, stacctype] = result;
[address, datasize DIV 8, stacctype];
if t != 31 then
    X[t] = ZeroExtend(data, 32);[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: 2018-09-13T13:25:04

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  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | (1)| (1)| (1)| (1)| (1)| 0  | (1)| (1)| (1)| (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 elsize = 8 << (integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32; // 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 << (integer s); // ignored by all loads and store-release;
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 "Rt" field.
Operation

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
dbits(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 = case memop of
  MemOp_STORE:
    data = X[t];
    Mem[address, dbytes, address, dbytes, acctype] = data;
  when
  MemOp_LOAD:
    data = ZeroExtend(data, regsize);
    X[t] = MemAccType_LIMITEDORDERED[MemOp_LOAD][address, dbytes, acctype];
  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.
```
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)

<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>1</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>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>Rn</td>
</tr>
</tbody>
</table>

size  L  Rs  o0  Rt2

No offset

LDLARB <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

```
bites(64) address;
bites(8) data;
bites(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 = case memop of
    when MemOp_STORE
        data = X[t];
    MemOp_LOAD
        data = Mem[address, 1, address, dbytes, acctype] = data;

when
    data = MemAccType_LIMITEDORDERED(MemOp_LOAD)(address, dbytes, acctype);
    X[t] = ZeroExtend(data, 32); //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  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

No offset

LDLARH <Wt>, [<Xn|SP>(,#0)]

integer n = UInt(Rn);
integer t  = UInt(Rt);  // ignored by load/store single register
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

```c
bits(64) address;
Bits(16) data;
Bits(datasize) data;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);
if n == 31 then
  CheckSPAlignment();
  address = SP[0];
else
  address = X[n];
data = case memop of
    when MemOp_STORE
      data = X[t];
      Mem[x+address, 2, address, dbytes, acctype] = data;
    when
      data = MemAccType_LIMITEDORDERED(MemOp_LOAD)(address, dbytes, acctype);
  X[t] = ZeroExtend(data, 32); (data, regsize);
```
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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.

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>}]

// Empty
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);

 if opc == '11' then UNDEFINED;
 integer scale = 2 + 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;
bites(datasize) data1;
bites(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = memop IN {MemOp LOAD, MemOp LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31);
  SetNotTagCheckedInstruction(is_load_store && t == 31 && !wback);

if t == t2 then if memop ==
  Memop LOAD t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LDPOWERLAP);
    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];

address = address + offset;
if ! postindex then
  address = address + offset;

data1 = case memop of
  when Memop STORE
    data1 = V[t];
    data2 = V[t2];
    Mem[address, dbytes, dbytes, acctype] = data1; AccType_VECSTREAM;
    data2 = Mem[address + dbytes, dbytes, acctype] = data2;
  when Memop LOAD
    data1 = Mem[address, dbytes, dbytes, acctype] = data1;
    data2 = Mem[address + dbytes, dbytes, acctype] = data2;
  if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[t2] = data2;[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: 2018-09-13T13:25Z

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

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>32-bit (opc == 00)</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>

LDNP <Wt1>, <Wt2>, [<Xn|SP>|, #<imm>]

64-bit (opc == 10)

LDNP <Xt1>, <Xt2>, [<Xn|SP>|, #<imm>]

// Empty.
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

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
AccType accctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UNDEFINED;
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_LOAD IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store & n == 31);
    SetNotTagCheckedInstruction(is_load_store && t = t & l1back);

if t == t2 then 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];

address = address + offset;
if ! postindex then
    address = address + offset;

data1 = 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, dbytes, address + 0, dbytes, acctype] = data1;
        AccType_STREAM[];
    data2 = address + dbytes, dbytes, acctype = data2;
    when MemOp_LOAD
        data1 = Mem[address+dbytes, dbyte, address + 0, dbyte, acctype];
        data2 = AccType_STREAM[];
    if rt_unknown then
        data1 = bits(datasize) UNKNOWN;
        data2 = bits(datasize) UNKNOWN; address = address + offset;
        if wback then
        if postindex then
            address = address + offset;
        if n == 31 then
            X[t] = data1;
        else
            X[t2] = data2;

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

| LDP |<St1>, <St2>, [<Xn|SP>], #<imm> |
|-----|---------------------------------|
| opc | 1 0 1 1 0 0 1 1 |
| imm7| Rtl |
| Rn  | Rtl |

#### 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

| LDP |<St1>, <St2>, [<Xn|SP>], #<imm> |
|-----|---------------------------------|
| opc | 1 0 1 1 0 1 1 |
| imm7| Rtl |
| Rn  | Rtl |

#### 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

| LDP |<St1>, <St2>, [<Xn|SP>], #<imm> |
|-----|---------------------------------|
| opc | 1 0 1 1 0 1 0 1 |
| imm7| Rtl |
| Rn  | Rtl |

### 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>!
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

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer accType = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
b bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```
Operation

```plaintext
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 LOAD, MemOp LOADIN};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if t == t2 then
  if memop == MemOp LOAD then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_known = TRUE;  // result is UNKNOWN
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
else
  address = X[n];

if !postindex then
  address = address + offset;

  case memop of
    when MemOp_STORE
      data1 = V[t];
      data2 = V[t2];
      Mem[address, dbytes, {address + 0 distance dbytes, acctype} = data1; AccType_VEC];
      data2 = {address + dbytes, dbytes, acctype} = data2;
    when MemOp_LOAD
      data1 = Mem[address+dbytes, dbytes, {address + 0 distance dbytes, acctype};
      data2 = {address + dbytes, dbytes, acctype};
    if rt_known then
      data1 = bits(dsize) UNKNOWN;
      data2 = bits(dsize) UNKNOWN;
    if rt_known then
      data1 = bits(dsize) UNKNOWN;
      data2 = bits(dsize) 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.
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</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>0</td>
<td>1</td>
<td>0</td>
<td>0</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>

opc

32-bit (opc == 00)

LDP <Wt1>, <Wt2>, [Xn|SP], #<imm>

boolean wback = TRUE;
boolean postindex = TRUE;

64-bit (opc == 10)

LDP <Xt1>, <Xt2>, [Xn|SP], #<imm>

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>x</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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)

LDP <Wt1>, <Wt2>, [Xn|SP], #<imm>!

64-bit (opc == 10)

LDP <Xt1>, <Xt2>, [Xn|SP], #<imm>!

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>x</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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)

LDP <Wt1>, <Wt2>, [Xn|SP]{, #<imm>}

64-bit (opc == 10)

LDP <Xt1>, <Xt2>, [Xn|SP]{, #<imm>}

boolean wback = TRUE;
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>` 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);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + (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;
bis(64) offset = LSL(SignExtend(imm7, 64), scale);
```

LDP

Page 441
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_LOAD, MemOp_STORE} && n == 31 && !wback;
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

boolean wb_unknown = FALSE;
if 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 wbback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF undefined;
    when Constraint_NOP EndOfInstruction();

if t == t2 then
  if memop == MemOp_STORE && (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;

data1 = case memop of
  when MemOp_STORE
    if rt unknown && t == n then
      data1 = bits(datasize) UNKNOWN;
    else
      data1 = X[t];
  when MemOp_LOAD
    data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype] = data1, AccType_NORMAL];
  data2 = [address + dbytes, dbytes, acctype] = data2;

  when MemOp_STORE
    if rt unknown then
      data1 = bits(datasize) UNKNOWN;
      data2 = bits(datasize) UNKNOWN;
  when MemOp_LOAD
    data2 = Mem[address+data2, dbytes, [address + 0, dbytes, acctype] = data2, AccType_NORMAL];
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
  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.
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  | 0  | 1  | 1  | imm7 | Rt2 | Rn | Rt |

opc L

#### Post-index

LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 1  | imm7 | Rt2 | Rn | Rt |

opc L

#### Pre-index

LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>]

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | imm7 | Rt2 | Rn | Rt |

opc L

#### Signed offset

LDPSW <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 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.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(RE2);
bits(64) offset = -RT2;
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if (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), 2);(imm7, 64, scale);
```
bits(64) address;
bv(32) data1;
bv(32) data2;

bits(datasize) data1;
bv(datasize) data2;

cconstant integer dbytes = datasize DIV 8;

boolean rt_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = boolean is load_store = memop IN {MemOp LOAD, MemOp STORE} SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then 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 wbback = FALSE;  // writeback is suppressed when Constraint_UNKNOWN wb_unknown = TRUE;  // writeback is UNKNOWN when Constraint_UNDEF UNDEFINED; when Constraint_NOP EndOfInstruction();

if t == t2 then if memop == MemOp STORE && t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
  assert c IN { Constraint_NONE, Constraint_UNDEFINED, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE;  // value stored is pre-writeback when Constraint_UNDEFINED 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
  if !postindex then
    address = address + offset;

data1 = 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, 4] [address + 0, dbytes, acctype] = data1; AccType_NORMALMem];
  data2 = [address + dbytes, dbytes, acctype] = data2;
  when MemOp_LOAD
    data1 = Mem[address+4, 4, [address + 0], dbytes, acctype];
    data2 = AccType_NORMALMem];
if rt_unknown then
data1 = bits(32) UNKNOWN;
data2 = bits(32) UNKNOWN; [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
    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.
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>
<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>

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>

```java
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-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>1</th>
<th>1</th>
<th>Rn</th>
<th>Rt</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>

Post-index , Pre-index and Unsigned offset
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

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm12</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDR (immediate, SIMD&FP)
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

```python
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
    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 AccType_VEC
            data = Mem[address, datasize DIV 8, AccType_VEC];
            V[t] = data;
        when MemOp_LOAD
            data = Mem[address, datasize DIV 8, AccType_VEC];
            Mem[address, datasize DIV 8, acctype] = data;
    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.

(old) htmldiff from- (new)
**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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | imm9 | 0  | 1  | Rn | | | | | | | | | | | | | | | | | | | | | | |

**32-bit (size == 10)**

```c
LDR <Wt>, [<Xn|SP>], #<simm>
```

**64-bit (size == 11)**

```c
LDR <Xt>, [<Xn|SP>], #<simm>
```

### 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  | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | imm9 | 1  | 1  | Rn | | | | | | | | | | | | | | | | | | | | | | |

**32-bit (size == 10)**

```c
LDR <Wt>, [<Xn|SP>], #<simm>
```

**64-bit (size == 11)**

```c
LDR <Xt>, [<Xn|SP>], #<simm>
```

### 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  | x  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | imm12 | | Rn | | | | | | | | | | | | | | | | | | | | | | | | |

```c
boolean wback = TRUE;
boolean postindex = TRUE;
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);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
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<1> == '1' then 32 else 64;
    signed = TRUE;
integer datasize = 8 << scale;
ifHaveMTEExt() then
    boolean is_load_store = memop IN {MemOp LOAD, MemOp LOAD, MemOp LOAD, MemOp STORE} // setNotTagCheckedInstruction(is_load_store && n == 31 && !wback);
    bits(64) address;
    bits(datasize) data;
    boolean wb_unknown = FALSE;
    boolean rt_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
        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 CheckSPA1ignment();
        else
            address = X[n];

    if !postindex then
        address = address + offset;
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
        MemOp_LOAD;
            data = Mem[address, datasize DIV 8, acctype] = data;
        when AccType_NORMAL 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(data, regsize);
            (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 (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.

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. |
| Xn/SP | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rm" 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. |
For the 8-bit variant: is the index extend specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>Option</th>
<th>Extend</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>

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>Extend</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>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<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>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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;
```
Operation

\[
\text{bits(64) offset = ExtendReg}(m, \text{extend_type}, \text{shift});
\]

\[
\text{if HaveMTEExt() then}
\]

\[
\text{boolean is_load_store = memop IN (MemOp STORE, MemOp LOAD);} ;
\]

\[
\text{SetNotTagCheckedInstruction(is_load_store \&\& n == 31);} ;
\]

\[
\text{CheckFPAdvSIMDEnabled64();}
\]

\[
\text{bits(64) address;}
\]

\[
\text{bits(datasize) data;}
\]

\[
\text{if n == 31 then}
\]

\[
\text{CheckSPAlignment();}
\]

\[
\text{else}
\]

\[
\text{address = X[n];}
\]

\[
\text{address = address + offset;}
\]

\[
\text{if ! postindex then}
\]

\[
\text{address = address + offset;}
\]

\[
\text{case memop of}
\]

\[
\text{when MemOp STORE}
\]

\[
\text{data = V[t];}
\]

\[
\text{Mem}[\text{address, datasize DIV 8}, \text{address, datasize DIV 8, acctype}] = data;
\]

\[
\text{when AccType_VEC}
\]

\[
\text{data = data;}
\]

\[
\text{when MemOp LOAD}
\]

\[
\text{data = Mem[\text{address, datasize DIV 8}, \text{address, datasize DIV 8, acctype}, AccType_VEC]; V[t] = data;}
\]

\[
\text{if wback then}
\]

\[
\text{if postindex then}
\]

\[
\text{address = address + offset;}
\]

\[
\text{if n == 31 then}
\]

\[
\text{[t] = address;}
\]

\[
\text{else}
\]

\[
\text{V[t] = data; X[n] = address;}
\]

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | Rm |    | option |   | S | 1 | 0 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size | opc |

32-bit (size == 10)

LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <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>

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);

integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;

AccType acctype = AccType.NORMAL;
MemOp memop;
boolean signed;

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;
  else
    if opc<0> == '1' then UNDEFINED;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

integer datasize = 8 << scale;
bits(64) offset = \texttt{ExtendReg}(m, \text{extend\_type}, \text{shift});

\textbf{if} HaveMTEExt() \textbf{then}
\begin{verbatim}
  boolean is_load_store = memop \in \{ \text{MemOp\_STORE}, \text{MemOp\_LOAD} \};
  SetNotTagCheckedInstruction(is_load_store \& n == 31); //
\end{verbatim}

\textbf{bits(64) address;}
\textbf{bits(datasize) data;}

\textbf{if} memop == \text{MemOp\_LOAD} \& wback \& n == t \& n != 31 \textbf{then}
\begin{verbatim}
  c = \text{ConstrainUnpredictable}(\text{Unpredictable\_WBOVERLAPLD});
  assert c \in \{ \text{Constraint\_WBSUPPRESS}, \text{Constraint\_UNKNOWN}, \text{Constraint\_UNDEF}, \text{Constraint\_NOP} \};
\end{verbatim}

\begin{verbatim}
\textbf{case} c \textbf{of}
\begin{verbatim}
  \text{when} \text{Constraint\_WBSUPPRESS} wback = \text{FALSE}; // writeback is suppressed
  \text{when} \text{Constraint\_UNKNOWN} \text{rt unknown} = \text{TRUE}; // value stored is UNKNOWN
  \text{when} \text{Constraint\_UNDEF} \text{UNDEFINED};
  \text{when} \text{Constraint\_NOP} \text{EndOfInstruction}();
\end{verbatim}
\end{verbatim}

\textbf{if} memop == \text{MemOp\_STORE} \& wback \& n == t \& n != 31 \textbf{then}
\begin{verbatim}
  c = \text{ConstrainUnpredictable}();
  SetNotTagCheckedInstruction(is_load_store \& n == 31);
\end{verbatim}

\textbf{bits(64) address;}
\textbf{bits(datasize) data;}

\textbf{if} n == 31 \textbf{then}
\begin{verbatim}
  \text{Unpredictable\_WBOVERLAPST};
  assert c \in \{ \text{Constraint\_NONE}, \text{Constraint\_UNKNOWN}, \text{Constraint\_UNDEF}, \text{Constraint\_NOP} \};
\end{verbatim}

\begin{verbatim}
\textbf{case} c \textbf{of}
\begin{verbatim}
  \text{when} \text{Constraint\_NONE} rt unknown = \text{FALSE}; // value stored is original value
  \text{when} \text{Constraint\_UNKNOWN} rt unknown = \text{TRUE}; // value stored is UNKNOWN
  \text{when} \text{Constraint\_UNDEF} \text{UNDEFINED};
  \text{when} \text{Constraint\_NOP} \text{EndOfInstruction}();
\end{verbatim}
\end{verbatim}

\textbf{if} n == 31 \textbf{then}
\begin{verbatim}
  \text{if} memop \neq \text{MemOp\__PREFETCH} \textbf{then} \text{CheckSPAignment}();
  \textbf{else}
  \text{address} = X[n];
\end{verbatim}

\textbf{address} = \text{address} + \text{offset};
\textbf{if} ! \text{postindex} \textbf{then}
\begin{verbatim}
  \text{address} = \text{address} + \text{offset};
\end{verbatim}

\textbf{data = case} memop \of
\begin{verbatim}
\text{when} \text{MemOp\_STORE}
\begin{verbatim}
  \textbf{if} rt unknown \textbf{then}
  \text{data} = \text{bits(datasize)} \text{UNKNOWN};
  \textbf{else}
  \text{data} = X[t];
\end{verbatim}

  \text{Mem(address, datasize \div 8, \text{acctype})} = \text{data};
\end{verbatim}

\begin{verbatim}
\text{when} \text{AccType\_NORMAL}\{\text{MemOp\_LOAD}\};\text{data} =
\text{Mem(address, datasize \div 8, \text{acctype});}
\begin{verbatim}
  \textbf{if} signed \textbf{then}
  \text{X[t]} = \text{SignExtend(data, regsize)};
  \textbf{else}
  \text{X[t]} = \text{ZeroExtend(data, regsize)};
\end{verbatim}

\end{verbatim}

\begin{verbatim}
\text{when} \text{MemOp\__PREFETCH\_Prefetch(address, t<4:0>)};
\begin{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}

\textbf{if} wback \textbf{then}
\begin{verbatim}
  \textbf{if} wb unknown \textbf{then}
  \text{address} = \text{bits(64)} \text{UNKNOWN};
  \textbf{elsif} \text{postindex} \textbf{then}
  \text{address} = \text{address} + \text{offset};
  \textbf{if} n == 31 \textbf{then}
  SP[] = \text{address};
\end{verbatim}

\textbf{if} signed \textbf{then}
\begin{verbatim}
\end{verbatim}
\end{verbatim}

\textbf{else}
\begin{verbatim}
\end{verbatim}
\end{verbatim}

\textbf{if} \text{wback unknown} \textbf{then}
\begin{verbatim}
\end{verbatim}
\end{verbatim}

\textbf{else if} \text{postindex} \textbf{then}
\begin{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}

else

X(data, regsize), [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 |1 |0 |0 |0 |M |S |1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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() then UNDEFINED;
if 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;
bv2int(64) offset = LSL(SignExtend(S10, 64), 3);(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> 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

bits(64) address;
bites(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

<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>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>
<tr>
<td>imm9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>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>
<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>size</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>0</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>imm9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>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>
<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>size</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>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>
</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>size</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pre-index

LDRB <Wt>, [Xn|SP], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = Integer scale = imm(size);
bias(64) offset = SignExtend(imm9, 64);

Pre-index

LDRB <Wt>, [Xn|SP], #<simm>

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = Integer scale = imm(size);
bias(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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>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>
</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>size</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unsigned offset

LDRB <Wt>, [Xn|SP], #<pimm>

boolean wback = FALSE;
boolean postindex = FALSE;
bias(64) offset = Integer scale = imm(size);
bias(64) offset = LSL(ZeroExtend(imm12, 64), 0);imm12, scale;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (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.
<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
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;
```

LDRB (immediate)
if HaveMTEExt() then
   boolean is_load_store = MemOp LOAD IN {MemOp STORE, MemOp LOAD};
   SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

   bits(64) address;
   bits(8) data;
   bits(datasize) data;

   boolean wb_unknown = FALSE;
   boolean rt_unknown = FALSE;

   if wback && n == t && n != 31 then
      c = MemOp LOAD 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 wbback = 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_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;

   data = case memop of
      when MemOp_STORE
         if rt_unknown then
            data = bits(datasize) UNKNOWN;
         else
            data = X[t];
            Mem[address, 1](address, datasize DIV 8, acctype) = data;
      when AccType_NORMAL 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(data, 32); (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.
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.

```
<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>0</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>option</td>
<td>S</td>
<td>1</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>
</tbody>
</table>
```

Extended register (option != 011)

LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]}

Shifted register (option == 011)

LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]}

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;
bits(64) offset = ExtendReg(m, extend_type, 0);

if HaveMTEExt() then

    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD} IN (1);
    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 = MemOp_LOADConstrainUnpredictable;
        SetNotTagCheckedInstruction(is_load_store && n == 31);

    bits(64) address;
    bits(8) data;

    if n == 31 then

        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();
        else
            address = X[n];
        address = address + offset;

        if ! postindex then
            address = address + offset;

        data = case memop of

            when MemOp_STORE

                if rt unknown then
                    data = bits(datasize) UNKNOWN.
                else
                    data = X[t];
            Mem[address, 1, address, datasize DIV 8, acctype] = data;

            when AccType_NORMAL[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_PREFETCHPrefetch(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

                else

                    address = X[t];
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

```
31 30 29 28 27 26 25 24 23 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 0 imm9 0 1 0 1 0
size opc
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = 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 1 0 imm9 1 1 0 1 0
size opc
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = 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 0 1 0 imm12 1 1 0 1 0
size opc
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
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

```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' 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 LOAD IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 & !wback);

    bits(64) address;
    Bits(16) data;
    Bits(datasize) data;

    boolean wb_unknown = FALSE;
    boolean rt_unknown = FALSE;

    if wback && n == t && n != 31 then
        c = if memop == MemOp LOAD && wback && n == t && n != 31 then
                ConstraintUnpredictable(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
        if memop == MemOp_STORE && wback && n == t && n != 31 then
            c = ConstraintUnpredictable(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();
        else
            address = X[n];

    if !postindex then
        address = address + offset;

    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, 2, address, datasize DIV 8, acctype] = data;
        when AccType_NORMAL MemOp_LOAD
            Mem[address, address, datasize DIV 8, acctype];
            if signed then
                X[t] = SignExtend(data, regsize);    // if signed
            else
                X[t] = ZeroExtend(data, regsize);  // if signed
        when MemOp_PREFETCH Prefetch(data, 32); (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.
**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](#).

![Encoding](image)

**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><code>&lt;extend&gt;</code></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><strong>S</strong></th>
<th><code>&lt;amount&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

LDRH `<Wt>, [<Xn|SP>, (<Wm>|<Xm>)], <extend> {<amount>}`

```plaintext
if option<1> == '0' then UNDEFINED; // sub-word index
boolean wback = 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 1 else 0; integer shift = if S == '1' then scale else 0;
```
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;
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    boolean is_load_store = boolean is_load_store = memop \in { MemOp_STORE, MemOp_LOAD } \in \{
        SetNotTagCheckedInstruction(is_load_store \&\& n = 31 \&\& \neg 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 rt unknown = FALSE; // value stored is original value
            when Constraint_UNDEF rt unknown = TRUE; // value stored is UNKNOWN
            when Constraint_NOP EndOfInstruction();
    
    if memop = MemOp_STORE \&\& wback \&\& n = t \&\& n != 31 then
        c = MemOp_LOADConstrainUnpredictable();
        SetNotTagCheckedInstruction(is_load_store \&\& n = 31);
    
    bits(64) address;
    bits(16) data;

    if n = 31 then
        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 \neq MemOp_PREFETCH then CheckSPAlignment();
        else
            address = X[n];

        address = address + offset;
        if \neg postindex then
            address = address + offset;

        data = case memop of
            when MemOp_STORE
                if rt unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = X[t];
                    Mem(address, 2, \{address, datasize \div 8, acctype\} = data);
            when AccType_NORMAL\{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;
        
        
    

LDRH (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.
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  | imm9 | 0 | 1 | Rn | Rt |

size opc

32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<simm>

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = 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  | imm9 | 1 | 1 | Rn | Rt |

size opc

32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<simm>!

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<simm>!

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = 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  | imm12 | 0 | 1 | Rn | Rt |

size opc
32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>{, #<pimm>}

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>{, #<pimm>}

boolean wback = FALSE;
boolean postindex = FALSE;

bits(64) offset = Integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0); (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 = 32;
  signed = FALSE;
else // sign-extending load
  memop = if size == '11' then UNDEFINED;
else
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

integer datasize = 8 << scale;
Operation
if \( \text{HaveMTEExt}() \) then
  boolean \( \text{is\_load\_store} = \text{memop IN} \{ \text{MemOp\_STORE}, \text{MemOp\_LOAD} \} \);
  SetNotTagCheckedInstruction(\( \text{is\_load\_store} \&\& \text{n == 31} \&\& \text{!wback} \);

bits(64) address;
bits(8) data;
bits(datasize) data;

boolean \( \text{wb\_unknown} = \text{FALSE} \);
boolean \( \text{rt\_unknown} = \text{FALSE} \);

if \( \text{memop == MemOp\_LOAD} \&\& \text{wback} \&\& \text{n == t} \&\& \text{n != 31} \) then
  \( \text{c} = \text{ConstrainUnpredictable} (\text{Unpredictable\_WBOVERLAPLD}) \);
  assert \( \text{c IN} \{ \text{Constraint\_WBSUPPRESS}, \text{Constraint\_UNKNOWN}, \text{Constraint\_UNDEF}, \text{Constraint\_NOP} \} \);
  case \( \text{c} \) of
    when \( \text{Constraint\_WBSUPPRESS} \) \( \text{wback} = \text{FALSE} \) then \( \text{writeback is suppressed} \);
    when \( \text{Constraint\_UNKNOWN} \) \( \text{wb\_unknown} = \text{TRUE} \) then \( \text{writeback is UNKNOWN} \);
    when \( \text{Constraint\_UNDEF} \) \( \text{UNDEFINED} \);
    when \( \text{Constraint\_NOP} \) \( \text{EndOfInstruction}() \);
  endcase;

if \( \text{memop == MemOp\_STORE} \&\& \text{wback} \&\& \text{n == t} \&\& \text{n != 31} \) then
  \( \text{c} = \text{ConstrainUnpredictable} (\text{Unpredictable\_WBOVERLAPST}) \);
  assert \( \text{c IN} \{ \text{Constraint\_NONE}, \text{Constraint\_UNKNOWN}, \text{Constraint\_UNDEF}, \text{Constraint\_NOP} \} \);
  case \( \text{c} \) of
    when \( \text{Constraint\_NONE} \) \( \text{rt\_unknown} = \text{FALSE} \) then \( \text{value stored is original value} \);
    when \( \text{Constraint\_UNKNOWN} \) \( \text{rt\_unknown} = \text{TRUE} \) then \( \text{value stored is UNKNOWN} \);
    when \( \text{Constraint\_UNDEF} \) \( \text{UNDEFINED} \);
    when \( \text{Constraint\_NOP} \) \( \text{EndOfInstruction}() \);
  endcase;

if \( \text{n == 31} \) then
  if \( \text{memop != MemOp\_PREFETCH} \) then \( \text{CheckSPAlignment}() \);
  address = \( \text{SP}[\text{n}] \);
else
  address = \( \text{X}[\text{n}] \);

if !\( \text{postindex} \) then
  \( \text{address = address + offset} \);

case \( \text{memop of} \)
  when \( \text{MemOp\_STORE} \) if \( \text{rt\_unknown} \) then
    \( \text{data = bits(8) UNKNOWN} \);
    \( \text{data = bits(datasize) UNKNOWN} \);
    \( \text{Mem[address, 1, address, datasize DIV 8, acc\_type] = data} \);
    when \( \text{AccType\_NORMAL} \) = data;
  when \( \text{MemOp\_LOAD} \) \( \text{data = Mem[address, 1, AccType\_NORMAL]} \);
    \( \text{address, datasize DIV 8, acc\_type] = data} \);
    if \( \text{signed} \) then
      \( \text{X[t] = SignExtend(data, reg\_size)} \);
    else
      \( \text{X[t] = ZeroExtend(data, reg\_size)} \);
    when \( \text{MemOp\_PREFETCH\_Prefetch(address, t<4:0>} \);

if \( \text{wb\_unknown} \) then
  \( \text{address = bits(64) UNKNOWN} \);
elsif \( \text{postindex} \) then
  \( \text{address = address + offset} \);
if \( \text{n == 31} \) then
  \( \text{SP[\text{n}] = address} \);
else
  \( \text{X[\text{n}] = address} \);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**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 *Load/Store addressing modes*.

<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>
</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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
</tr>
<tr>
<td>option</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Rn</td>
</tr>
<tr>
<td>Rt</td>
</tr>
</tbody>
</table>

**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>}]`

```
if option<1> == '0' then UNDEFINED; // sub-word index
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;             // subword index
ExtendType extend_type = DecodeRegExtend(option);  // 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;
        regsize = 32;
        signed = FALSE;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
else
    // sign-extending load
    memop = if opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
bits(64) offset = ExtendReg(m, extend_type, 0);  
\{m, extend_type,_shift\}.
if HaveMTEExt() then
    boolean Is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);
bits(64) address;
bits(8) data;
bits(datasize) data;
if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, address, datasize DIV 8, acctype] = data;
    when AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 1, address, datasize DIV 8, acctype];
        if signed then AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH PREFIX = data;
    if wback then
        if wb unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = 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

<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>1</td>
<td>x</td>
<td>0</td>
<td>imm9</td>
<td>0</td>
<td>1</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>
</tr>
</tbody>
</table>

- **size**: opc

#### 32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>], #<simm>

#### 64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>], #<simm>

```java
boolean wback = TRUE;
boolean postindex = TRUE;
bias(64) offset = Integer.scale << size;
bias(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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>imm9</td>
<td>1</td>
<td>1</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>
</tr>
</tbody>
</table>

- **size**: opc

#### 32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>], #<simm>!

#### 64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>], #<simm>!

```java
boolean wback = TRUE;
boolean postindex = FALSE;
bias(64) offset = Integer.scale << size;
bias(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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>imm12</td>
<td>0</td>
<td>1</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>
</tr>
</tbody>
</table>

- **size**: opc
32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>{, #<pimm>]

64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>{, #<pimm>]

boolean wback = FALSE;
broadcast postindex = FALSE;
bits(64) offset = Integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);((imm12, 64), scale);

For information about the CONTRAINTED 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 = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = if size == '11' then MemOp_LOAD else MemOp_STORE;
  regsize = 64;
  signed = TRUE;
else
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;
integer datasize = 8 << scale;
Operation

LDRSH (immediate)
if  
  if HaveMTEExt() then 
  boolean is_load_store = memop IN [MemOp_STORE, MemOp_LOAD]; 
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback); 
bits(64) address; 
Bits(16) data; 
Bits(32) 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 
  if ! postindex then 
    address = address + offset; 
  case memop of 
    when MemOp_STORE 
      if rt_unknown then 
        data = bits(16) UNKNOWN; 
        data = bits(data) UNKNOWN; 
      else 
        data = X[t]; 
        Mem[address, 2, address, datasize DIV 8, acctype] = data; 
      when AccType_NORMAL] = data; 
    when MemOp LOAD 
      data = Mem[address, 2, AccType_NORMAL]; 
      address, datasize DIV 8, acctype]; 
      if signed then 
        X[t] = SignExtend(data, regsize)); 
      else 
        X[t] = ZeroExtend(data, regsize)); 
      when MemOp_PREFETCHPrefetch(address, t<4:0>); 
if wb_unknown 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>])]

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>001</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 = \texttt{UInt}(Rn);
integer t = \texttt{UInt}(Rt);
integer m = \texttt{UInt}(Rm);

	exttt{AccType} acctype = \texttt{AccType\_NORMAL};
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};
    regsize = 32;
    signed = FALSE;
else
  // sign-extending load
  memop = if opc<0> == '1' then \texttt{UNDEFINED};
  else
    // sign-extending load
    memop = \texttt{MemOp\_LOAD};
    regsize = if opc<0> == '1' then 32 else 64;
    signed = \texttt{TRUE};
else
  // sign-extending load
  memop = if opc<0> == '1' then \texttt{UNDEFINED};
  else
    // sign-extending load
    memop = \texttt{MemOp\_LOAD};
    regsize = if opc<0> == '1' then 32 else 64;
    signed = \texttt{TRUE};

integer datsize = 8 << scale;
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);
    SetNotTagCheckedInstruction(is_load_store & n == 31 & !wback);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        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;
            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;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then
            CheckSPAlignment();
        else
            address = X[n];

        address = address + offset;

    case memop of
        when MemOp_STORE
            data = if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
            Mem[address, 2, address, datasize DIV 8, acctype] = data;
        when AccType_NORMAL] = data;

        when MemOp_LOAD
            data = Mem[address, 2, address, datasize DIV 8, acctype];
            if signed then AccType_NORMAL];
            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;
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 1 0 0 0 imm9 0 1 Rn Rt
```

size opc

```
LDRSW <Xt>, [<Xn|SP>], #<imm>
```

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = Integer.scale = SignExtend(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 1 0 0 0 imm9 1 1 Rn Rt
```

size opc

```
LDRSW <Xt>, [<Xn|SP>], #<imm>]
```

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = Integer.scale = SignExtend(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 0 1 0 0 imm12 1 Rn Rt
```

size opc

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}
```

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = Integer.scale = SignExtend(size);
bits(64) offset = LSZ(ZeroExtend(imm12, 64), 2);(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.
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 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

**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;
```
Operation

if HaveMTEExt() then

boolean is_load_store = MemOp LOAD IN {MemOp STORE, MemOp LOAD};
SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(32) data;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_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
    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = 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
        if !postindex then
            address = SP[];
        else
            address = X[n];

        if !postindex then
            address = address + offset;

        data = case memop of
            when MemOp_STORE
                if rt unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = X[t];
                    Mem[address, 4];[address, datasize DIV 8, acctype] = data;
            when AccType_NORMAL(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;X[t] = Fetch(data, 64); (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.

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

| 31 30 29 28 27 26 25 24 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 | 0 | 0 | imm19 | | Rt | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 

opc

Literal

LDRSW <Xt>, <label>

integer t = UInt(Rt);
bits(64) offset;
offset = 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;
  when '11' 
    signed = TRUE;
    memop = MemOp_PREFETCH;
  offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<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

bits(64) address = PC[] + offset;
bits(32) data;
bits(size*8) data;
data = case memop of
  when MemOp_LOAD
    data = Mem[address, 4];
  when MemOp_PREFETCH
    Prefetch(data, 64);
  if signed then
    X[t] = SignExtend(data, 64);
  else
    X[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.
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.

### 64-bit

LDRSW \(<Xt>, [<Xn|\text{SP}], (<Wm>|<Xm>)\{, <\text{extend}> [<\text{amount}>]\}\)

```plaintext
if option<1> == '0' then UNDEFINED; // sub-word index
boolean wback = FALSE;
integer scale =
if option<1> == '0' then UNDEFINED;             // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 else 0; 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.
- \(<Xn|\text{SP}>\)  
  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.
- \(<\text{extend}>\)  
  Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when \(<\text{amount}>\) is omitted. \(<\text{option}>\) encoded in "option":

<table>
<thead>
<tr>
<th>(&lt;\text{option}&gt;)</th>
<th>(&lt;\text{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>

- \(<\text{amount}>\)  
  Is the index shift amount, optional only when \(<\text{extend}>\) is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>(&lt;S&gt;)</th>
<th>(&lt;\text{amount}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</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;
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); // uback)

bits(64) address;
bits(datasize) data;

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 wback = FALSE; // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  end case;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = MemOp_LOADConstrainUnpredictable();
  SetNotTagCheckedInstruction(is_load_store && n == 31);

bits(64) address;
bits(32) data;

if n == 31 then
  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();
  else
    address = X[n];
    address = address + offset;
    if ! postindex then
      address = address + offset;
  end if;
  data = case memop of
    when MemOp_STORE
      if rt unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
        Mem(address, 4, address, datasize DIV 8, acctype) = data;
      end if;
    when AccType_NORMAL MemOp_LOAD];
      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 ub unknown then
      address = bits(64) UNKNOWN;
    elsif postindex then
      address = address + offset;
    else if n == 31 then
      SP[] = 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Opc |

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

```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];

result = data OR value;
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.
**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.**

```plaintext
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   | 1   | 0   | 0   | Rn  | Rt  |     |     |     |     |     |     |     |     |     |     |     |     |     |

size  opc

**LDSETB (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;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs) + 2;
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.

<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>STSETB, STSETLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bv(8) value;
bv(8) data;
bv(8) result;
bv(datasize) value;
bv(datasize) data;
bv(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, 1, ldacctype];
[address, datasize DIV 8, ldacctype];

result = data OR value;# 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_SHL  =>  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SHLB  =>  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMR  =>  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMRA  =>  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMRB  =>  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMRB  =>  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMRB  =>  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMRB  =>  result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMRB  =>  result = if SInt(data) > SInt(value) then data else value;

Mem[address, 1, stacctype] = result;
[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, 32));
```

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, LDSETALH, 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, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETH and LDSETALH store to memory with release semantics.
- LDSETH 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 STSETH, STSETLH.

### LDSETH (A == 0 && R == 0)

LDSETH `<Ws>`, `<Wt>`, `[<Xn|SP>]`

### LDSETAH (A == 1 && R == 0)

LDSETAH `<Ws>`, `<Wt>`, `[<Xn|SP>]`

### LDSETALH (A == 1 && R == 1)

LDSETALH `<Ws>`, `<Wt>`, `[<Xn|SP>]`

### LDSETH (A == 0 && R == 1)

LDSETH `<Ws>`, `<Wt>`, `[<Xn|SP>]`

### LDSETLH (A == 0 && R == 1)

LDSETLH `<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' && 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>STSETH, STSETLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
b(16) value;
b(16) data;
b(16) result;
b(datasize) value;
b(datasize) data;
b(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.

result = data OR value; case op of
when
    MemAtomicOp_ADD result = data + value;
    MemAtomicOp_BIC result = data AND NOT(value);
    MemAtomicOp_EOR result = data EOR value;
    MemAtomicOp_ORR result = data OR value;
    MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
    MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
    MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;

Mem[address, 2, stacctype] = result;
[address, data size DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, 32); [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 STS MAX, STS MAXL.

### Integer

(ARMv8.1)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | 1 | 1 | 1 | 0 | 0 | A | R | 1 | Rs | 0 | 1 | 0 | 0 | 0 | Rn | Rt |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | x |   |   |   |   |   |   |   |   |   |   |   |   |   |
| size | opc |
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

<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.
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;
b bits(datasize) value;
b bits(datasize) data;
b 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];

result = if case op of
    when MemAtomicOp_ADD then data + value;
    when MemAtomicOp_BIC then data AND NOT(value);
    when MemAtomicOp_EOR then data EOR value;
    when MemAtomicOp_ORR then data OR value;
    when MemAtomicOp_SMAX then if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN then if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX then if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN then 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.
LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

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)

<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>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### LDSMAXB (A == 1 && R == 0)

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

#### LDSMAXAB (A == 1 && R == 1)

LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]

#### LDSMAXB (A == 0 && R == 0)

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

#### LDSMAXLB (A == 0 && R == 1)

LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]

```c
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); if size then
integer datasize = 8 <<
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDERED_ATOMICRW else AccType_ATOMIC_RW;
AccType stacctype = if R == '1' then AccType_ORDERED_ATOMICRW else AccType_ATOMIC_RW;
MemAtomicOp nop;
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

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
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, 1, ldacctype];
result = if 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 data else value;
  Mem[address, 1, stacctype] = result;
  [address, datasize DIV 8, stacctype] = result;
if t != 31 then
  X[t] = ZeroExtend(data, 32), (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 | Rt |

size opc

LDSMAXH (A == 1 && R == 0)

LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]

LDSMAXAH (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 <<

if datasize == 64 then 64 else 32;

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.

<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(16) value;
bits(16) data;
bits(16) result;
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.
result = if 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, 2, stacctype] = result;
[address, datasize DIV 8, stacctype] = result;

if t != 31 then
  X[t] = ZeroExtend(data, 32);[data, receive];
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMIN, LDSMINA, LDSMINAL, LDSMINL

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  | x  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |

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_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>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];

result = if case op of
    MemAtomicOp_ADD         result = data + value;
    MemAtomicOp_BIC         result = data AND NOT(value);
    MemAtomicOp_EOR         result = data EOR value;
    MemAtomicOp_ORR         result = data OR value;
    MemAtomicOp_SMAX        result = if SInt(data) > SInt(value) then data else value;
    MemAtomicOp_SMIN        result = if SInt(data) > SInt(value) then value else data;
    MemAtomicOp_UMAX        result = if UInt(data) > UInt(value) then data else value;
    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 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.

### LDSMINB

    Integer

    (ARMv8.1)

#### LDSMINB (A == 1 && R == 0)

```text
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMINB (A == 0 && R == 0)

```text
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMINB (A == 0 && R == 1)

```text
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMINB (A == 1 && R == 1)

```text
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```
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 == '1111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
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, 1, ldacctype];
address, datasize DIV 8, ldacctype];
result = if case op of
    MemAtomicOp_ADD result = data + value;
    MemAtomicOp_BIC result = data AND NOT(value);
    MemAtomicOp_EOR result = data EOR value;
    MemAtomicOp_ORR result = data OR value;
    MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
    MemAtomicOp_SMIN result = if SInt(data) < SInt(value) then value else data;
    MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then value else data;
    MemAtomicOp_UMIN result = if UInt(data) < UInt(value) then data else value;
Mem[address, 1, stacctype] = result;
address, datasize DIV 8, stacctype] = result;
if t != 31 then
    X[t] = ZeroExtend(data, 32); [data, receive];
```

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: 2018-09-13T13:25
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</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>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rs</td>
<td>Rn</td>
</tr>
<tr>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>

#### 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>]

```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' && 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_ADDA;
    when '001' op = MemAtomicOp_BICA;
    when '010' op = MemAtomicOp_EORA;
    when '011' op = MemAtomicOp_ORRA;
    when '100' op = MemAtomicOp_SMAXA;
    when '101' op = MemAtomicOp_SMINA;
    when '110' op = MemAtomicOp_UMAXA;
    when '111' op = MemAtomicOp_UMINA;
```
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>STSMINH, STSMINLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
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, 2, ldacctype];
result = if case op of
  MemAtomicOp_ADD result = data + value;
  MemAtomicOp_BIC result = data AND NOT(value);
  MemAtomicOp_EOR result = data EOR value;
  MemAtomicOp_ORR result = data OR value;
  MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
  MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
  MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
  MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
Mem[address, 2, stacctype] = result;
result = if case op of
  MemAtomicOp_ADD result = data + value;
  MemAtomicOp_BIC result = data AND NOT(value);
  MemAtomicOp_EOR result = data EOR value;
  MemAtomicOp_ORR result = data OR value;
  MemAtomicOp_SMAX result = if SInt(data) > SInt(value) then data else value;
  MemAtomicOp_SMIN result = if SInt(data) > SInt(value) then value else data;
  MemAtomicOp_UMAX result = if UInt(data) > UInt(value) then data else value;
  MemAtomicOp_UMIN result = if UInt(data) > UInt(value) then value else data;
Mem[address, 2, stacctype] = result;
if t != 31 then
  X[t] = ZeroExtend(data, 32); // 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.
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.

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;
int regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
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_LOAD 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 n == 31 then
    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();
else
    address = X[n];

address = address + offset;
if !postindex then
    address = address + offset;
data = 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;
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.

 Unscaled offset

LDTRB <Wt>, [<Xn|SP>{, #<simm>}]

`bits(64) offset = 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
ingress n = UInt(Rn);
ingress 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
    // sign-extending load
    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;
    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 = MemOp_LOAD ConstrainUnpredictable;
        SetNotTagCheckedInstruction(is_load_store & n == 31);
        if n == 31 then
            SP[] = address;
            bits(8) data;
            if n == 31 then
                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();
        else
            address = [n];
        address = address + offset;
        if !postindex then
            address = address + offset;
        data = case memop of
            when MemOp_STORE
                if rt_unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = [t];
                    Mem[address, 1, acctype][address, datasize DIV 8, acctype] = data;
            when MemOp_PREFETCH
                data = Mem[address, datasize DIV 8, acctype];
                if signed then
                    [t] = SignExtend(data, regsize);
                else
                    [t] = ZeroExtend(data, regsize);
                when MemOp_PREFETCHPrefetch(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(data, 32); [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

```assembly
LDTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

```assembly
define boolean wback = FALSE;
define boolean postindex = FALSE;
define integer scale = UInt(size);
define 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

```assembly
ingter n = UInt(Rn);
ingter 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';
```

```assembly
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL; memop = MemOp_STORE;
boolean signed;
ingter regsize;
```

```assembly
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;
```

LDTRH
if HaveMTEExt() then

boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD}
SetNotTagCheckedInstruction(is_load_store && n == 31)

bits(64) address;
bits(datasize) data;

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 = MemOp_LOADConstrainUnpredictable();
SetNotTagCheckedInstruction(is_load_store & n == 31);

bits(64) address;
bits(16) data;

if n == 31 then

    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();
else
    address = X[n];

address = address + offset;
if !postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, acctype];[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(data, 32); 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.

31 30 29 28 27 26 25 24 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>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0 0 0 1 x 0</td>
<td>imm9 1 0</td>
</tr>
</tbody>
</table>

32-bit (opc == 11)

LDTRSB <Wt>, [<Xn|SP>{, #<simm>}]

64-bit (opc == 10)

LDTRSB <Xt>, [<Xn|SP>{, #<simm>}]

| Bits(64) offset = | 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) && HaveVirtExt() && 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 = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = if size == '11' then MemOp_LOAD else MemOp_STORE;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;
  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 \{MemOp\_STORE, MemOp\_LOAD\};
  SetNotTagCheckedInstruction(is_load_store \&\& n == 31);
  SetNotTagCheckedInstruction(is_load_store \&\& n == 31 \&\& \!wback);
  bits(64) address;
  bits(8) data;
  bits(datasize) data;
  if n == 31 then
    if memop \!\! 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 wb\_unknown = \FALSE; // writeback is suppressed
        when Constraint\_UNKNOWN wb\_unknown = \TRUE; // writeback is UNKNOWN
        when Constraint\_UNDEF UNDEFINED;
        when Constraint\_NOP EndOfInstruction();
    end_case
  end_if
  if n == 31 then
    if memop \!= MemOp\_PREFETCH then
      CheckSPAlignment();
    end_if
    address = SP[];
  end_if
  else
    address = X[n];
  end_case
  address = address + offset;
  if \! postindex then
    address = address + offset;
  end_if
  case memop of
    when MemOp\_STORE
      data = \!
        if rt\_unknown then
          data = bits(datasize) UNKNOWN;
        end_if
        else
          data = X[t];
          Mem[address, 1, acctype] = data;
          [address, datasize DIV 8, acctype] = data;
        end_case
      when MemOp\_LOAD
        data = Mem[address, 1, acctype];
        [address, datasize DIV 8, acctype] = data;
        if signed then
          X[t] = SignExtend(data, regsize);
        else
          X[t] = ZeroExtend(data, regsize);
        end_if
      end_case
    when MemOp\_PREFETCH
      Prefetch(address, t<4:0>);
    end_case
  end_case
  if \! wback then
    if wb\_unknown then
      address = bits(64) UNKNOWN;
    end_if
    elsif postindex then
      address = address + offset;
    end_if
    if n == 31 then
      SP[] = address;
    end_if
    else
      X[(address, t<4:0>);[n] = address;
end_case
\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.
LDTRSH

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | opc |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | x  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | imm9  | 1  | 0  | Rn  | Rt  |

32-bit (opc == 11)

LDTRSH <Wt>, [<Xn|SP>{, #<simm>}

64-bit (opc == 10)

LDTRSH <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.
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 = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then UNDEFINED
        else MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
else
    // sign-extending load
    memop = if opc<0> == '1' then UNDEFINED
        else MemOp_LOAD;
    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);
  bits(64) address;
  bits(16) data;
  bits(datasize) data;
  if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
    else
      address = X[n];
      address = address + offset;
    end if
  end if
  case memop of
    when MemOp STORE
      data = if rt unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
      end if
      Mem[address, 2, acctype] = data;
      [address, datasize DIV 8, acctype] = data;
    when MemOp LOAD
      data = Mem[address, 2, acctype];
      [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[(address, t<4:0>)[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.
LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extend 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.

```
31 30 29 28 27 26 25 24 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   opc
1 0 1 1 1 0 0 0 1 0 0 | imm9 1 0 | Rn | Rt
```

Unscaled offset

LDTRSW <Xt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = 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);

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;
```
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;

    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 rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_NOP EndOfInstruction();
    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = MemOp_LOADConstrainUnpredictable();
        SetNotTagCheckedInstruction(is_load_store && n == 31);

    bits(64) address;
    bits(32) data;

    if n == 31 then
        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 rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_NOP EndOfInstruction();
    if n == 31 then
        if memop == MemOp_PREFETCH then CheckSPAlignment();
    else
        address = SP[];

        address = address + offset;
        if !postindex then
            address = address + offset;

        data = case memop of
            when MemOp_STORE
                if rt_unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = X[t];
                    Mem[address, 4, acctype];[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);
                EndOfInstruction();
            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;
        }
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  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 0  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size | opc
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 LDUMAXA (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;
iinteger t = UInt(Rt);
iinteger n = UInt(Rn);
iinteger s = UInt(Rs);

iinteger datasize = 8 << UInt(size);
iinteger 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>STUMAX, STUMAXL</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];
result = if 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  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 0  | 0  | Rn | 0  | Rt |

size opc

LDUMAXB (A == 1 & R == 0)

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

LDUMAXAB (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 <<

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.

<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>STUMAXB, STMUXLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```c
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
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, 1, ldacctype];
address, datasize DIV 8, ldacctype];
result = if 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, 1, stacctype] = result;
address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, 32),4(data, receive);
```

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 0 A R 1 Rs 0 1 1 0 0 0 Rn Rt

size opc

LDUMAXAH (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 != '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>STUMAXH, STUMAXLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
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, 2, ldacctype];
[address, datasize DIV 8, ldacctype];
result = if 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, 2, stacctype] = result;
[address, datasize DIV 8, stacctype] = result;
if t != 31 then
  X[t] = ZeroExtend(data, 32);4(data, receive);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 1  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
</table>

---

**LDUMIN, LDUMINA, LDUMINAL, LDUMINL**  
Page 572
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.
<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>STUMIN, STUMINL</td>
<td>A == '0' &amp;&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
    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];
result = if case op of
    when MemAtomicOp_ADD then data + value;
    when MemAtomicOp_BIC then data AND NOT(value);
    when MemAtomicOp_EOR then data EOR value;
    when MemAtomicOp_ORR then data OR value;
    when MemAtomicOp_SMAX then if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN then if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX then if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN then 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.
LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

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, STUMINLB.

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  | A | R | 1 | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| size | opc |

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>, [\(<Xn|SP>\)]

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
    "000" op = MemAtomicOp_ADD;
    "001" op = MemAtomicOp_BIC;
    "010" op = MemAtomicOp_EOR;
    "011" op = MemAtomicOp_ORR;
    "100" op = MemAtomicOp_SMAX;
    "101" op = MemAtomicOp_SMIN;
    "110" op = MemAtomicOP_UMAX;
    "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>STUMINB, STUMINLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
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, 1, ldacctype];
result, datasize DIV 8, ldacctype];
result = if 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, 1, stacctype] = result;
result, datasize DIV 8, stacctype] = result;
if t != 31 then
    X[t] = ZeroExtend(data, 32);4(data, receive);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

(\textit{old}) \quad \textit{htmldiff from} - \quad (\textit{new})
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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 1  | 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    | 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>]

if !HaveAtomicExt() then UNDEFINED;
  integer t = UInt(Rt);
  integer n = UInt(Rn);
  integer s = UInt(Rs);#Rs;
  integer datasize = 8 << #opc;
  integer regsize = if datasize == 64 then 64 else 32);
  AccType ldacctype = if A == '1' &amp; 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 case;
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; Rt == '11111'</td>
</tr>
</tbody>
</table>

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;
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, 2, ldacctype];
data, datatype DIV 8, ldacctype];
result = if 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, 2, stacctype] = result;
data, datatype DIV 8, stacctype] = result;
if t != 31 then
  X[t] = ZeroExtend(data, 32), data, receive);
```

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>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.

<S> 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);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;
if ! postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    data = V[t];
    Mem[address, datasize DIV 8, address, datasize DIV 8, acctype] = data;
  when AccType_VEC
    data = Mem[address, datasize DIV 8, address, datasize DIV 8, acctype];
    V[acc][t] = data;
  when MemOp_LOAD
    data = Mem[address, datasize DIV 8, address, datasize DIV 8, acctype];
    AccType_VEC][t] = data;
  if ! check then
    if postindex then
      address = address + offset;
    if n == 31 then
      address = address + offset;
    else
      address = address + offset;

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: 2018-09-13T13:25
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

### 32-bit (size == 10)

LDUR <Wt>, [<Xn|SP>], #<simm>

### 64-bit (size == 11)

LDUR <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

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<0> == '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} IN (1);
    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 = MemOp_LOADConstrainUnpredictable();
        SetNotTagCheckedInstruction(is_load_store & n == 31);

    bits(64) address;
    bits(datasize) data;

    if n == 31 then
        Unpredictable_WBOVERLAPST);
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE rtUnknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN rtUnknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop == MemOp_PREFETCH then CheckSPAEmitter();
        else
            address = X[n];

        address = address + offset;
        if !postindex then
            address = address + offset;

        data = case memop of
            when MemOp_STORE
                if rt unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = X[t];
            Mem[(address, datasize DIV 8), (address, datasize DIV 8, acctype) = data];
            when AccType_NORMAL[Memop LOAD];
                data = Mem[(address, datasize DIV 8, acctype), (address, datasize DIV 8, acctype), signed then
                    X[t] = SignExtend(data, regsize);
                else
                    X[t] = ZeroExtend(data, regsize);
            when Memop_PREFETCHPrefetch(address, t<4:0>);

        if wback then
            address = bits(64) UNKNOWN;
        else if postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
else
\* (data, regsize) \{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>}

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

\[
\begin{align*}
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } t &= \text{UInt}(Rt); \\
\text{AccType } accType &= \text{AccType_NORMAL}; \\
\text{MemOp } memop &= \text{MemOp_STORE}; \\
\text{boolean } signed &= \text{FALSE}; \\
\text{integer } regsize &= \text{if size == '11' then 64 else 32}; \\
\text{integer } datasize &= 8 \ll \text{scale}; \\
\end{align*}
\]
if HaveMTEExt() then
    boolean is_load_store = MemOp_STORE, MemOp_LOAD IN (1).
    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_UNDEFINED, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP);
        case c of
            when Constraint_UNDEFINED EndOfInstruction();
            when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
            when Constraint_UNDEF EndOfInstruction();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        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 EndOfInstruction();
            when Constraint_NOP EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then
            CheckSPAlignment();
        else
            address = X[n];

        address = address + offset;
        if !postindex then
            address = address + offset;

        data = case memop of
            when MemOp_STORE
                if rt_unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = X[t];
                end;
            when AccType_NORMAL[MemOp_LOAD];data =
                Mem[address, datasize DIV 8, acctype] = data;
            when MemOp_PREFETCH
                X[t] = HighExtend(data, regsize);
            when MemOp_PREFETCHPrefetch(address, t<4:0>);

        if wback then
            address = bits(64) UNKNOWN;
        elseif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
else

\( x_{(data, 32)}; \text{ln} \rightarrow \text{address} \)

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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>

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 accotype = AccType_NORMAL;
MemOp memop = MemOp_STORE;
boolean signed = FALSE;
integer regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;

if opc<0> == '1' 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_LOAD, MemOp_STORE };
    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

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable(MemOp_LOAD);
        SetNotTagCheckedInstruction(is_load_store & n == 31);

        bits(64) address;
        bits(16) data;

        if n == 31 then
            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
    end

    if n == 31 then
        if memop != MemOp_PREFETCH then
            CheckSPAlignment();
        else
            address = X[n];
        end
        address = address + offset;
    if !postindex then
        address = address + offset;
    data = case memop of
        when MemOp_STORE
            if rt unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
                Mem[address, 2] = address, datatype DIV 8, acctype] = data;
        when AccType_NORMAL[MemOp_LOAD]; data =
            Mem[address, datatype 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 if wback then
            if wb unknown then
                address = bits(64) UNKNOWN;
            elseif postindex then
                address = address + offset;
            if n == 31 then
                SP[] = address;
else
{data, 32};[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*.

32-bit (opc == 11)

LDURSB <Wt>, [<Xn|SP>{, #<simm>}]  

64-bit (opc == 10)

LDURSB <Xt>, [<Xn|SP>{, #<simm>}]  

```plaintext
bits(64) offset = SignExtend(imm9, 64);
integer scale = UInt(size);
boolean wback = FALSE;
boolean postindex = FALSE;
```

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_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;
    regsize = 32;
    signed = FALSE;
else
  // sign-extending load
  memop = if opc<0> == '1' then UNDEFINED;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;
  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 \texttt{HaveMTEExt()} then
  boolean \texttt{is\_load\_store} = \texttt{memop IN \{MemOp\_STORE, MemOp\_LOAD\}};

  \texttt{SetNotTagCheckedInstruction(is \_load\_store && n \&= 31)};

  \texttt{SetNotTagCheckedInstruction(is \_load\_store && n \&= 31 \& \texttt{!wback})};

  \texttt{bits(64) address; bits(8) data; bits(datasize) data;}

  if \texttt{n == 31} then
    if \texttt{memop != MemOp\_LOAD} && \texttt{wback \&\& n == t \&\& n 1 31 then}
      \texttt{c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPLD);}
      \texttt{assert c IN \{Constraint\_WSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDER, Constraint\_NOP\};}
      \texttt{case c of}
      \texttt{when Constraint\_WSUPPRESS \texttt{wback} = FALSE; \texttt{// writeback is suppressed}}
      \texttt{when Constraint\_UNKNOWN \texttt{wb\_unknown} = TRUE; \texttt{// writeback is UNKNOWN}}
      \texttt{when Constraint\_UNDER \texttt{UNDEFINED};}
      \texttt{when Constraint\_NOP \texttt{EndOfInstruction}();}

    if \texttt{memop != MemOp\_STORE} && \texttt{wback \&\& n == t \&\& n != 31 then}
      \texttt{c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPST);}
      \texttt{assert c IN \{Constraint\_NONE, Constraint\_UNKNOWN, Constraint\_UNDER, Constraint\_NOP\};}
      \texttt{case c of}
      \texttt{when Constraint\_NONE \texttt{rt\_unknown = FALSE; \texttt{// value stored is original value}}}
      \texttt{when Constraint\_UNKNOWN \texttt{rt\_unknown = TRUE; \texttt{// value stored is UNKNOWN}}}
      \texttt{when Constraint\_UNDER \texttt{UNDEFINED};}
      \texttt{when Constraint\_NOP \texttt{EndOfInstruction}();}

  if \texttt{n == 31} then
    if \texttt{memop != MemOp\_PREFETCH} then
      \texttt{CheckSPAlignment();}
    else
      \texttt{address = SP[];}

      \texttt{address = address + offset;}
      \texttt{if ! postindex then}
        \texttt{address = address + offset;}

      \texttt{case memop of}
      \texttt{when MemOp\_STORE}
        \texttt{data = if rt\_unknown then}
          \texttt{data = bits(datasize) UNKNOWN;}
        \texttt{else}
          \texttt{data = X[t];}
        \texttt{Mem[address, 1, address, datasize DIV 8, acctype] = data;}
        \texttt{when AccType\_NORMAL] = data;}

      \texttt{when MemOp\_LOAD}
        \texttt{data = Mem[address, 1, address, datasize DIV 8, acctype];}
        \texttt{if signed then AccType\_NORMAL];}

        \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 \texttt{wback} then
      \texttt{if wb\_unknown then}
        \texttt{address = bits(64) UNKNOWN;}
      \texttt{elsif postindex then}
        \texttt{address = address + offset;}
      \texttt{if \texttt{n == 31 then}
        \texttt{SP[]} = address;
      \texttt{else}
        \texttt{X[address, t<4:0>][]} = 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.

32-bit (opc == 11)

LDURSH <Wt>, [<Xn|SP>{, #<simm>}

64-bit (opc == 10)

LDURSH <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.
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;
    regsize = 32;
    signed = FALSE;
else
  // sign-extending load
  memop = if opc<0> == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
if size == '10' && opc<0> == '1' then UNDEFINED;
regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;
to integer datasize = 8 << scale;
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store & n == 31);
    SetNotTagCheckedInstruction(is_load_store & n == 31 & !wback);

    bits(64) address;
    bits(16) data;
    bits(datasize) data;

    if n == 31 then
        if memop != MemOp_PREFETCH then
            CheckSPAlignment();
        else
            address = X[n];
        end
        address = address + offset;
        if ! postindex then
            address = address + offset;
        end
        case memop of
            when MemOp STORE
                data = if rt unknown then
                    bits(datasize) UNKNOWN;
                else
                    X[t];
                end
                Mem[address, 2, address, datasize DIV 8, acctype] = data;
                when AccType NORMAL] = data;
            when MemOp LOAD
                data = Mem[address, 2, address, datasize DIV 8, acctype];
                if signed then
                    AccType NORMAL];
                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 wb unknown then
                            address = bits(64) UNKNOWN;
                        else
                            address = address + offset;
                        end
                        if n == 31 then
                            SP[] = address;
                        else
                            X[address, t<4:0>];[n] = address;
                    end
                end
            end
        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.
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.

<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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm9</td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td>Rt</td>
</tr>
</tbody>
</table>

Unscaled offset

LDURSW \(<Xt>, [<Xn|SP>|#, \#<simm>\]\

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.

\(<\text{simm}>\) 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
  // sign-extending load
  memop = if opc<0> == '1' then MemOp_PREFETCH;
  if opc<0> == '1' then UNDEFINED;
  else
    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
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD} IN {;
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 = MemOp_LOADConstrainUnpredictable();
    SetNotTagCheckedInstruction(is_load_store & n == 31);

bits(64) address;
bits(32) data;

if n == 31 then
    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();
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;
data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
            Mem[address, 4], address, datasize DIV 8, acctype] = data;
        when AccType_NORMAL[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_PREFETCHPrefetch(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;
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*.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  |

32-bit (sz == 0)

LDXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

LDXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer elsize = 32 << UInt(sz); // ignored by load/store single register
integer datasize = elsize * 2; // 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 regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;

For information about the CONTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDXP*.

**Assembler Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Wt1&gt;</td>
<td>Is the 32-bit name of the first general-purpose register to be transferred, encoded in the &quot;Rt&quot; field.</td>
</tr>
<tr>
<td>&lt;Wt2&gt;</td>
<td>Is the 32-bit name of the second general-purpose register to be transferred, encoded in the &quot;Rt2&quot; field.</td>
</tr>
<tr>
<td>&lt;Xt1&gt;</td>
<td>Is the 64-bit name of the first general-purpose register to be transferred, encoded in the &quot;Rt&quot; field.</td>
</tr>
<tr>
<td>&lt;Xt2&gt;</td>
<td>Is the 64-bit name of the second general-purpose register to be transferred, encoded in the &quot;Rt2&quot; field.</td>
</tr>
<tr>
<td>&lt;Xn</td>
<td>SP&gt;</td>
</tr>
</tbody>
</table>
bits(64) address;
bv(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if HaveMTEExt() then
  boolean is_load_store = -- boolean is_load_store = memop IN {MemOp LOAD, MemOp_STORE} SetNotTagCheckedInstruction(is_load_store && n == 31);
if t == t2 then 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 rt_unknown = UNDEFINED;
    when Constraint_NOP EndOfInstruction();
if n == 31 then if memop == MemOp_STORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    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 EndOfInstruction();
if s == n && n != 31 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 EndOfInstruction();
if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];
// 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. 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];
    end if;
    bit status = '1';
  end if;
// 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
  Mem[address, dbytes, acctype] = data;
end if;
...
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 rt_unknown then
  // ConstrainedUNPREDICTABLE case
  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, address, dbytes, acctype];
      if AccType_ATOMIC;
        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 when
      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;
        end when
      AArch64.Abort(address, AArch64.AlignmentFault((acctype, iswrite, secondstage));
      Mem[address + 0, 8, acctype];
      X[t] = Mem[address, 8, address + 8, 8, acctype];
      X[t2] = Mem[address+8, 8, AArch64.AlignmentFault((acctype, iswrite, secondstage));
      X[t] = Mem[address, 8, address + 8, 8, acctype];
    end when
  end when
end when

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 elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32; // 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); // ignored by load/store single register
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 "Rt" field.
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if HaveMTEExt() then
  boolean is_load_store = boolean is_load_store = memop IN { MemOp_STORE, MemOp LOAD } IN (4);
  SetNotTagCheckedInstruction(is_load_store && n == 3i);
if memop == MemOp LOAD && pair && s == t2 then
  Constraint c = ConstraintUnpredictable(Unpredictable адресное перекрытие);
  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 MemOp LOAD Constraint;
  SetNotTagCheckedInstruction(is_load_store && n == 3i);
if n == 3i then
  ConstraintUnpredictable(Unpredictable DATAOVERLAP);
  assert c IN { Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP };
  case c of
    when Constraint_UNKNOWN rt unknown = TRUE; // store UNKNOWN value
    when Constraint_UNDEF rt unknown = FALSE; // store original value
    when Constraint_NOP EndOfInstruction();
  if s == n || n != 3i then
    ConstraintUnpredictable(Unpredictable BASEOVERLAP);
    assert c IN { Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP };
    case c of
      when Constraint_UNKNOWN rt unknown = FALSE; // address is UNKNOWN
      when Constraint_UNDEF rt unknown = FALSE; // address is original base
      when Constraint_NOP EndOfInstruction();
  if n == 3i then
    CheckSPAlignment();
    address = SP[];
  else rt unknown then
    address = bits(64) UNKNOWN;
  else
    address = X[n];
  // 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.
  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);
data = 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, address, dbytes, acctype],
    if () 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.
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*.

No offset

```
LDXRB <Wt>, [<Xn|SP>]{,#0}
```

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>Rs</th>
<th>o0</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</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 == '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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = true = memop IN { MemOp_STORE, MemOp_LOAD ];
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp_LOAD if 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 MemOp_LOAD; Constraint;
  SetNotTagCheckedInstruction(is_load_store & n == 31);

if n == 31 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 n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

// 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 dbyte-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.

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, 1);

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, 1, address, dbytes, acctype];
    if () 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
  end
  // 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 = MemAccType_ATOMIC_BigEndian[address, dbytes, acctype];
    X[t] = ZeroExtend(data, 32);;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.
LDXRH

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

LDXRH <Wt>, [<Xn|SP>|, #0]}

integer n = UInt(Rn);
integer t = UInt(Rt); // ignored by load/store single register
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(16) data;
bits(datasize) datasize;
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} IN {MemOp_STORE, MemOp_LOAD}
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstraintUnpredictableUnpredictable(DATAOVERLAP);
    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();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then MemOp_LOAD(Constraint);
    SetNotTagCheckedInstruction(is_load_store & n == 31);

if n == 31 then
    Constraint c = ConstraintUnpredictableUnpredictable(DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEFINED, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN then rt_unknown = TRUE;  // store UNKNOWN value
        when Constraint_UNDEFINED then rt_unknown = FALSE;  // store original value
        when Constraint_NOP then EndOfInstruction();

if memop == MemOp_STORE then
    if n == n && n != 31 then
        Constraint c = ConstraintUnpredictableUnpredictable(BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEFINED, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN then rt_unknown = TRUE;  // address is UNKNOWN
            when Constraint_UNDEFINED then rt_unknown = FALSE;  // address is original base
            when Constraint_NOP then EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = $SP[];
else
    if rn_unknown then
        address = bits(64) UNKNOWN;
    else
        address = X[n];
    // 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. 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.
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
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 MUL.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | 0  | Rm | 0  | Ra | Rn | Rd |
| o0 |

32-bit (sf == 0)

MADD <Wd>, <Wn>, <Wm>, <Wa>

64-bit (sf == 1)

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

<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 addend, 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 addend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>
Operation

\[
\begin{align*}
\text{bits(destsize) operand1} &= \text{bits(datasize) operand1} = X[n]; \\
\text{bits(destsize) operand2} &= \text{bits(datasize) operand2} = X[m]; \\
\text{bits(destsize) operand3} &= X[a]; \\
\text{integer result} = \text{if sub_op then} \\
\text{result} &= \text{UInt(operand3)} + (\text{operand3} - (\text{ UInt(operand1) * UInt(operand2)}); \\
\text{else} \\
\text{result} &= \text{UInt(operand3)} + (\text{UInt(operand1) * UInt(operand2)}); \\
X[d] &= \text{result<destsize-1:0>};
\end{align*}
\]

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.
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.

Vector

MLA <Vd>.<T>, <Vn>.<T>, <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 = 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”: 

The assembler symbols provide a mapping for encoding the various fields in the instruction format. The vector elements and their operations are specified in a structured manner, allowing for efficient execution in SIMD&FP contexts.

MLA (by element)
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsz) 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.

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 datasize = if Q == '1' then 128 else 64;
integer elements = datasize 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”:

<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) 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)*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 (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.

### Assembler Symbols

<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;T&gt;</td>
<td>Is an arrangement specifier, encoded in “size:Q”:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>Q</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<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>

<table>
<thead>
<tr>
<th>&lt;Vn&gt;</th>
<th>Is the name of the first SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Vm&gt;</td>
<td>Is the name of the second SIMD&amp;FP source register, encoded in “size:M:Rm”:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>&lt;Vm&gt;</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------</td>
</tr>
<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.

<table>
<thead>
<tr>
<th>&lt;Ts&gt;</th>
<th>Is an element size specifier, encoded in “size”:</th>
</tr>
</thead>
</table>

| 00 | RESERVED |
| 01 | 0:Rm     |
| 10 | M:Rm     |
| 11 | RESERVED |
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsize) 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>;
  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  | size | 1  | 1  | 0  | 1  | 0  | 1  | Rd |
| U  |    |   |    |    |    |    | Rm  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Three registers of the same type

$$\text{MLS } <Vd>.<T>, <Vn>.<T>, <Vm>.<T>$$

```plaintext
ingterger d = UInt(Rd);
ingterger n = UInt(Rn);
ingterger m = UInt(Rm);
if size == '11' then UNDEFINED;
ingterger esize = 8 << UInt(size);
ingterger datasize = if Q == '1' then 128 else 64;
ingterger elements = datasize DIV esize;
boolea 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

```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) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*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.

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

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

<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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>hw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>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)

MOVK <Wd>, #<imm>[, LSL #<shift>]

64-bit (sf == 1)

MOVK <Xd>, #<imm>[, LSL #<shift>]

```plaintext
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = bits(16) imm = imm16;
integer pos; MoveWideOp opcode;

case opc of
  when '00' opcode = MoveWideOp_N[d];
  when '10' opcode = MoveWideOp_Z[d];
  when '11' opcode = MoveWideOp_K[d];
  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.

**Operation**

```plaintext
bits(datasize) result;
result = if opcode == MoveWideOp_K then
  result = X[d];
else
  result = Zeros();
result<pos+15:pos> = imm;
if opcode == MoveWideOp_N[d];
result<pos+15:pos> = imm16;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.
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)

```
31 30 29 28 27 26 25 24 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 1 hw imm16 1 Rd
opc
```

MOVN <Wd>, #<imm>{, LSL #<shift>}

64-bit (sf == 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
sf 0 0 1 0 0 1 0 1 hw imm16 1 Rd
opc
```

MOVN <Xd>, #<imm>{, LSL #<shift>}

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = bits(16) imm = imm16;
integer pos;
integer 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>64-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>64-bit</td>
<td><img src="imm16" alt="IsZero" /> &amp; hw != '00'</td>
<td></td>
</tr>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>32-bit</td>
<td><img src="imm16" alt="IsZero" /> &amp; hw != '00' &amp; <img src="imm16" alt="IsOnes" /></td>
<td></td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;

result = if opcode == MoveWideOp_K then
  result = X[d];
else
  result = zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N();
result<pos+15:pos> = imm16;
result = NOT(result);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</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</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)

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 pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = 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>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (wide immediate)</td>
<td>!(IsZero(imm16) &amp; hw != '00')</td>
</tr>
</tbody>
</table>
Operation

```plaintext
bits(datasize) result;

result = if opcode == MoveWideOp_K then
           result = X[d];
         else
           result = zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N();
result<pos+15:pos> = imm16;
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|  0|  1|  1|  o0| op1|  CRn|  CRm|  op2|  Rt|

System

MRS <Xt>, (systemreg|S<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></th>
<th>&lt;op0&gt;</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></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(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2));
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.

### System MSR <pstatefield>, #<imm>

```assembly
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');
```

```assembly
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 a 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”: 

![MSR diagram](MSR_diagram.png)
<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>SP Sel</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>11x</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>DAIF Set</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 = CRm<0>;
    PSTATE.SSBS = operand<0>;
when PSTATEField_SP
    PSTATE.SP = CRm<0>;
    PSTATE.SP = operand<0>;
when PSTATEField_DAIFSet
    PSTATE.D = PSTATE.D OR CRm<3>;
    PSTATE.A = PSTATE.A OR CRm<2>;
    PSTATE.I = PSTATE.I OR CRm<1>;
    PSTATE.F = PSTATE.F OR CRm<0>;
    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(CRm<3>);
    PSTATE.A = PSTATE.A AND NOT(CRm<2>);
    PSTATE.I = PSTATE.I AND NOT(CRm<1>);
    PSTATE.F = PSTATE.F AND NOT(CRm<0>);
    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 = CRm<0>;
    PSTATE.PAN = operand<0>;
when PSTATEField_UAO
    PSTATE.UAO = CRm<0>;
    PSTATE.UAO = operand<0>;
when PSTATEField_DIT
    PSTATE.DIT = CRm<0>;
    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.

System

MSR (\texttt{<systemreg>|S<op0>_<op1>_Cn_<Cm>_<op2>}), \texttt{<Xt>}

\begin{verbatim}
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');
\end{verbatim}

Assembler Symbols

\texttt{<systemreg>} \hspace{1em} Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".

The System register names are defined in \textit{'AArch64 System Registers' in the System Register XML}.

\texttt{<op0>} \hspace{1em} Is an unsigned immediate, encoded in "o0":

\begin{center}
\begin{tabular}{|c|c|}
\hline
o0 & \texttt{<op0>} \\
\hline
0 & 2 \\
1 & 3 \\
\hline
\end{tabular}
\end{center}

\texttt{<op1>} \hspace{1em} Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

\texttt{<Cn>} \hspace{1em} Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

\texttt{<Cm>} \hspace{1em} Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

\texttt{<op2>} \hspace{1em} Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

\texttt{<Xt>} \hspace{1em} Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

\begin{verbatim}
if read then
  \texttt{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, \texttt{X[t]});
\end{verbatim}
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.

<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>Rm</th>
<th>1</th>
<th>Ra</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
</tbody>
</table>

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

MSUB \(<Wd>, <Wn>, <Wm>, <Wa>\)

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

MSUB \(<Xd>, <Xn>, <Xm>, <Xa>\)

```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

- \(<Wd>\) 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>MNEG</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>
Operation

\[
\begin{align*}
\text{bits}(\text{destsize}) \text{ operand1} &= \text{bits}(\text{datasize}) \text{ operand1} \quad X[n]; \\
\text{bits}(\text{destsize}) \text{ operand2} &= \text{bits}(\text{datasize}) \text{ operand2} \quad X[m]; \\
\text{bits}(\text{destsize}) \text{ operand3} &= X[a]; \\
\end{align*}
\]

integer result;

\[
\begin{align*}
\text{result} &= \text{if sub_op} \text{ then} \\
&\quad \text{result} = \text{UInt}(\text{operand3}) - (\text{UInt}(\text{operand1}) \times \text{UInt}(\text{operand2})); \\
\text{else} &\quad \text{result} = \text{UInt}(\text{operand3}) + (\text{UInt}(\text{operand1}) \times \text{UInt}(\text{operand2})); \\
\end{align*}
\]

\[
X[d] = \text{result}<\text{destsize}-1:0>;
\]

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 `CPACR_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  | 1  | size| L  | M  | Rm | 1  | 0  | 0  | H  | 0  |    |    |    |    |    |    |    |    |    |    |    |

### Vector

MUL <Vd>..<T>, <Vn>..<T>, <Vm>..<Ts>[<index>]

```plaintext
def integer idxdsiz = if H == '1' then 128 else 64;
def integer index;
def bit Rmhi;
def case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

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

def integer esize = 8 << UInt(size);
def integer datar = if Q == '1' then 128 else 64;
def integer elements = datar 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>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 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>;
  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 (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.

<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>1</th>
<th>1</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<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>
</tr>
</tbody>
</table>

Three registers of the same type

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

```plaintext
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

```c
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)*4element1) + 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.
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.

| 31 30 29 28 27 26 25 24 23 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 0 0 a b c | cmode 0 1 d e f g h | Rd |
| op                          | 16-bit shifted immediate (cmode == 10x0) |
| MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>} |

32-bit shifted immediate (cmode == 0xx0)

MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}

32-bit shifting ones (cmode == 110x)

MVNI <Vd>.<T>, #<imm8>, MSL #<amount>

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 '0xx01' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = ImmediateOp_MOVI;
  when '0xx11' operation = ImmediateOp_BIC;
  when '0xx01' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '10x01' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = ImmediateOp_MOVI;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '1111x' operation = ImmediateOp_MOVI;
  when '1111x' operation = ImmediateOp_MOVI;

imm64 = AdvSIMDEncodeImm(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 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”:

MVNI
<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>":

<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.
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.

System

```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, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
when '0100 xx0'
            op = SystemHintOp_BTI;
            BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
otherwise EndOfInstruction();// Empty,4;                     // Instruction executes as
```
Operation

// do nothing
when SystemHintOp_YIELD Hint Yield;

when SystemHintOp_WFE
  if !InterruptPending() then
    ClearEventRegister();
  else
    if PSTATE.EI == $1 then
      // Check for traps described by the OS which may be EL1 or EL2.
      AArch64.CheckForWFETrap($11, TRUE);
      if EL2Enabled() && PSTATE.EI IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFETrap($12, TRUE);
      if HaveEL(EL2) && PSTATE.EI == $13 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFETrap($13, TRUE);
      WaitForEvent();
    when SystemHintOp_WFI
      if !InterruptPending() then
        if PSTATE.EI == $10 then
          // Check for traps described by the OS which may be EL1 or EL2.
          AArch64.CheckForWFETrap($11, FALSE);
          if EL2Enabled() && PSTATE.EI IN {EL0, EL1} && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFETrap($12, FALSE);
          if HaveEL(EL2) && PSTATE.EI == $13 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFETrap($13, FALSE);
        WaitForInterrupt();
    when SystemHintOp_SEV SendEvent();
    when SystemHintOp_SEVL SendEventLocal();
    when SystemHintOp_ESB SynchronizeErrors();
      AArch64.ESBOperation();
      if EL2Enabled() && PSTATE.EI IN {EL0, EL1} then
        AArch64.vESBOperation();
      TakeUnmaskedErrorInterrupts();
    when SystemHintOp_PSB ProfilingSynchronizationBarrier();
    when SystemHintOp_TSB TraceSynchronizationBarrier();
    when SystemHintOp_CSDB ConsumptionOfSpeculativeDataBarrier();
    when SystemHintOp_BTI BTypeNext = '00';
  otherwise // do nothing

Operational information

If PSTATE.DIT is 1:

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

Three registers of the same type

ORN <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 "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

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bites(datasize) operand2 = V[m];
bites(datasize) result;

operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);
result = operand1 OR operand2; case op of
  when LogicalOp_AND
    result = operand1 AND operand2;
  when LogicalOp_ORR
    result = operand1 ORR operand2;
  when LogicalOp_XOR
    result = operand1 XOR 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.

32-bit (sf == 0)

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

64-bit (sf == 1)

ORN <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;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
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)4imm6;4imm6;
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>MVN</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

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

<table>
<thead>
<tr>
<th>Q</th>
<th>Rd</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>a b c x x x 1 0 1 d e f g h</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
<td>2S</td>
</tr>
</tbody>
</table>

### 16-bit (cmode == 10x1)

ORR <Vd>.<T>, #<imm>{, LSL #<amount>}

### 32-bit (cmode == 0x1)

ORR <Vd>.<T>, #<imm>{, LSL #<amount>}

integer rd = UInt(Rd);

integer datatype = if Q == '1' then 128 else 64;

bits(datatype) imm;

bits(64) imm64;

**ImmediateOp** operation;

```plaintext
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx10' operation = ImmediateOp_MVNI;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x10' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = ImmediateOp_MOVI;
  when '0000' operation = ImmediateOp_MOVI;
  when '0010' operation = ImmediateOp_MVNI;
  when '0011' operation = ImmediateOp_MOVI;
  when '1000' operation = ImmediateOp_MOVI;
  when '1001' operation = ImmediateOp_BIC;
  when '1010' operation = ImmediateOp_MVNI;
  when '1011' operation = ImmediateOp_BIC;
  when '1100' operation = ImmediateOp_MOVI;
  when '1101' operation = ImmediateOp_MOVI;
  when '1110' operation = ImmediateOp_MOVI;
  when '1111' operation = ImmediateOp_MOVI;

// 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, datatype 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".
For the 16-bit 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 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();
bite(datasize) operand;
bite(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).

<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>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 1 0 0 1 0 0 1 Rm 0 0 0 1 1 1 Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Three registers of the same type

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

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

integer datasize = if Q == '1' then 128 else 64;
integer esize = 8;
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”:
  - 0 8B
  - 1 16B
- `<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

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
better(result);
result = operand1 OR operand2; if invert then operand2 = NOT(operand2);

switch op of
  when LogicalOp_AND
       result = operand1 AND operand2;
  when LogicalOp_OR
       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)**.

### 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;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = 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;
   end;
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>
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;
result = operand1 OR imm;
if d == 31 then case op of
  when LogicalOp_AND
      result = operand1 AND operand2;
  when LogicalOp_ORR
      result = operand1 OR operand2;
  when LogicalOp_EOR
      result = operand1 EOR operand2;
end
if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
end
if d == 31 || !setflags then
  SP[] = result;
else
  X[d] = result;
end

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 (shifted register)

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)

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

64-bit (sf == 1)

ORR <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;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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);4imm6);
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>MOV (register)</td>
<td><code>shift == '00' &amp;&amp; imm6 == '000000' &amp;&amp; Rn == '11111'</code></td>
</tr>
</tbody>
</table>

### Operation

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

result = operand1 OR operand2; 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 XOR 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.
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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Z</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>
</tr>
</tbody>
</table>

PACIA (Z == 0)

PACIA \(<Xd>\), \(<Xn|SP>\)

PACIZA (Z == 1 & Rn == 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>0</td>
<td>1</td>
<td>1</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>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CRm | op2
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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | Rn | Rd |

**PACIB (Z == 0)**

PACIB <Xd>, <Xn|SP>

**PACIZB (Z == 1 && Rn == 11111)**

PACIZB <Xd>

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 1  | 0  | 0  | 0  | x  | 1  | 0  | 1  | x  | 1  | 1  | 1  | 1  | 1 | CRm | op2 |
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 HaveBTIEExt() then
            BTypeCompatible = BTypeCompatible_PACIBSP();
        end if
    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]);
end if

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

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.

Three registers of the same type

PMUL $<Vd>$, $<T>$, $<Vn>$, $<T>$, $<Vm>$, $<T>$

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>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 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)*4element1-4UInt(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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 | opc |

Unsigned offset

\[
\text{PRFM} \ (<\text{prfop}>|\#<\text{imm5}>), \ [<\text{Xn}|\text{SP}>\{, \ #<\text{pimm}>\}
\]

Assembler Symbols

\(<\text{prfop}>
\)

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 \(<\text{imm5}>
\).

\(<\text{imm5}>
\)

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 \(<\text{prfop}>
\).

\(<\text{Xn}|\text{SP}>
\)

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rt" field.

\(<\text{pimm}>
\)

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 \(<\text{pimm}>>8\).
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()
{
    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 = ConstrInUnpredictable(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 rt unknown = FALSE;       // value stored is original value
            when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrInUnpredictable(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 IN {ThenCheckAlignment()};
                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;
                    SetNotTagCheckedInstruction(is_load_store && n == 31);
                bits(64) address;

            if n == 31 then
                address = data = SPMem();
            else
                address = address, datasize DIV 8, acctype);
                if signed then X[n];
            address = address + offset;

            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;

    PRFM (immediate)
else

X(address, t<4:0>); \( n \) = 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 \texttt{PRFM} instruction is \textit{IMPLEMENTATION DEFINED}. For more information, see \textit{Prefetch memory}.

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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | imm19 |

\texttt{opc}

Literal

\texttt{PRFM (<prfop>|#<imm5>), <label>}

```plaintext
integer t = UInt(Rt);
offset = (Rt<<4); memop = MemOp_LOAD;
signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00' size = 4;
  when '01' size = 8;
  when '10' size = 4;
  when '11' signed = TRUE;
    memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

\texttt{<prfop>} Is the prefetch operation, defined as \texttt{<type><target><policy>}. \texttt{<type>} is one of:

- \texttt{PLD} Prefetch for load, encoded in the \texttt{"Rt<4:3>"} field as \texttt{0b00}.

- \texttt{PLI} Preload instructions, encoded in the \texttt{"Rt<4:3>"} field as \texttt{0b01}.

- \texttt{PST} Prefetch for store, encoded in the \texttt{"Rt<4:3>"} field as \texttt{0b10}.

\texttt{<target>} is one of:

- \texttt{L1} Level 1 cache, encoded in the \texttt{"Rt<2:1>"} field as \texttt{0b00}.

- \texttt{L2} Level 2 cache, encoded in the \texttt{"Rt<2:1>"} field as \texttt{0b01}.

- \texttt{L3} Level 3 cache, encoded in the \texttt{"Rt<2:1>"} field as \texttt{0b10}.

\texttt{<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

```c
bits(64) address = PC[] + offset;[] + 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: 2018-09-13T13:25:04

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

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>})

if option<1> == '0' then UNDEFINED;    // sub-word index
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 3 else 0;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 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 "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.
When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

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>

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
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;
    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;
```

integer datasize = 8 << scale;
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 & (t != 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 vb unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  end case;

if memop == MemOp_STORE & wback & n == t & (t != 31) then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  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 IN {then CheckSPAlignment();}
    address = SP[];
  else
    address = X[n];
  end if;

if ! postindex then
  address = address + offset;
end case;

if memop == MemOp_STORE & rt unknown then
  data = bits(datasize) UNKNOWN;
  else
    data = X[t];
    Mem[address, datasize DIV 8, acctype] = data;
end if;

SetNotTagCheckedInstruction(is_load_store & n == 31);

bits(64) address;

if n == 31 then
  address = data == SP[];
else
  address = address + offset;[t]
end if;

if signed then
  X[t] = SignExtend(data, regsize);
else
  X[t] = ZeroExtend(data, regsize);
end if;

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;
  end if;

if n == 31 then
else

X(address, t<4:0>)[n] = address;

PRFM (register)
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 PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

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
 size  opc

Unscaled offset

PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}

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 = \texttt{ UInt}(Rn);
integer t = \texttt{ UInt}(Rt);\texttt{ AccType acctype = AccType\_NORMAL};
\texttt{ MemOp memop};
boolean signed;
\texttt{ 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};
else

if opc[0] == '1' then \texttt{ UNDEFINED};
else

// sign-extending load
memop = \texttt{ MemOp\_LOAD};
if size == '10' && opc[0] == '1' then \texttt{ UNDEFINED};
regsize = if opc[0] == '1' then 32 else 64;
signed = TRUE;

\texttt{ 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 IN {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;
                SetNotTagCheckedInstruction(is_load_store && n == 31);

    bits(64) address;
    if n == 31 then
        address = data = %SPm[];
    else
        address = address, datasize DIV 8, acctype;
            if signed then %X[n];

    address = address + offset;[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;

    PRFM (unscaled offset)
else

X(address, t<4:0>);[n] = address.
PSB CSYNC

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>1</td>
</tr>
</tbody>
</table>
```

System

```
if !op;

SystemHintOp =
  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 '0001 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 '1001 xxx' \typeCompatible = \typeCompatible_BT() then(op2<2:1>);
    otherwise   EndOfInstruction();
```

PSB CSYNC
case op of
  when SystemHintOp_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($11, TRUE);
          if EL2Enabled() && PSTATE_EL IN ($10, $11) && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFXTrap($12, TRUE);
          if HaveEL($13) && PSTATE_EL == EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFXTrap($13, 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($11, FALSE);
          if EL2Enabled() && PSTATE_EL IN ($10, $11) && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFXTrap($12, FALSE);
          if HaveEL($13) && PSTATE_EL = EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFXTrap($13, FALSE);
        WaitForInterrupt();
    when SystemHintOp_SEV
        SendEvent();
    when SystemHintOp_SEVL
        SendEventLocal();
    when SystemHintOp_ESB
        SynchronizeErrors();
        AArch64.ESBOperation();
        if EL2Enabled() && PSTATE_EL IN ($10, $11) then AArch64.vESBOperation();
        TakeUnmaskedErrorInterrupts();
    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: 2018-09-13T13:28
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```c
System

PSSBB

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;// Empty.
  end;
```

PSSBB
case op of
when MemBarrierOp_DSBB, 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 ();
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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  |

RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D

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.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];

V[d] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));

Operational information

If PSTATE.DIT is 1:

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

• The response of this instruction to asynchronous exceptions does not vary based on:
  o The values of the data supplied in any of its registers.
  o 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.

Integer

```
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;
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;
```

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.
Operation

```c
bits(64) target = X[n];

case branch_type of
    when BranchType_INDIR (target, 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.
RETAA, RETAB

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
(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 1 0 1 1 1 1 0 0 0 0 1  M 1 1 1 1 1 1 1 1 1 1 1

   Z op  A  Rn  Rm
```

RETAA (M == 0)

```
RETAA
```

RETAB (M == 1)

```
RETAB

boolean use_key_a = (M == '0');
if !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 UNDEFINED;
        if n != 31 then UNDEFINED;
        n = 30;
        source_is_sp = TRUE;
```
Operation

bits(64) target = X[30];
bias(64) modifier = n;

case branch_type of
    when BranchType_INDIR
        if InGuardedPage then
            if n == 16 || n == 17 then
                BTypeNext = '01';
            else
                BTypeNext = '11';
            end;
        else
            BTypeNext = '11';
        end;
    when BranchType_INDCALL
        BTypeNext = '10';
    when BranchType_RET
        BTypeNext = '00';
if pac then
    bits(64) modifier = if source_is_sp then SP[];
if use_key_a then
    target = [] else [4][4];
    if use_key_a then
        target = AuthIA(target, modifier);
    else
        target = AuthIB(target, modifier);
end;
if branch_type == BranchType_INDCALL then X[30] = PC[] + 4;
BranchTo(target, BranchType_RET);/target, branch_type);
REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</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>
<th>0</th>
<th>0</th>
<th>1</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<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>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>
<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>
</tr>
</tbody>
</table>

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 == '1' 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.
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.

```
| 31 30 29 28 27 26 25 24 23 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 | 0 | 1 | 1 | 0 | Rn | Rd |
```

Vector

REV16 <Vd>.<T>, <Vn>.<T>

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

// size=esize:   B(0),  H(1),  S(1),  D(S)
define integer esize = 8 << UInt(size);
define integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
define 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;

define integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
endcase

define integer containers = datasize DIV container_size;
define 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.
REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd
opc

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.
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 CPACR_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

REV32 <Vd>.<T>, <Vn>.<T>

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)
binary(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>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</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
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.
REV32

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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>
</tr>
<tr>
<td>sf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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 |

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.
REV64

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.

Vector

REV64 <Vd>.<T>, <Vn>.<T>

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

// size=esize: B(0), H(1), S(1), D(4)
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”:

| size | Q | <T>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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.
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.
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| 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | Rn | 0  | mask |

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

```plaintext
bits(4) tmp;
bots(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.

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

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 CPACR_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

SABA \(<Vd>.<T>, <Vn>.<T>, <Vm>.<T>\)

\[
\begin{align*}
\text{integer } d & = \text{UInt}(Rd); \\
\text{integer } n & = \text{UInt}(Rn); \\
\text{integer } m & = \text{UInt}(Rm); \\
\text{if } size &= '11' \text{ then UNDEFINED;} \\
\text{integer } esize & = 8 << \text{UInt}(size); \\
\text{integer } datasize & = \text{if } Q &= '1' \text{ then 128 else 64;} \\
\text{integer } elements & = \text{datasize DIV } esize; \\
\text{boolean } unsigned & = (U == '1'); \\
\text{boolean } accumulate & = (ac == '1');
\end{align*}
\]

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>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

\[
\begin{align*}
\text{CheckFPAdvSIMDEnabled64();} \\
\text{bits(datasize) operand1 = } V[n]; \\
\text{bits(datasize) operand2 = } V[m]; \\
\text{bits(datasize) result;} \\
\text{integer element1;} \\
\text{integer element2;} \\
\text{bits(esize) absdiff;}
\end{align*}
\]

\[
\begin{align*}
\text{result} & = \text{if accumulate then } V[d] \text{ else } \text{Zeros}(); \\
\text{for } e & = 0 \text{ to elements-1} \\
\text{element1} & = \text{Int}(\text{Elem[operand1, e, esize], unsigned}); \\
\text{element2} & = \text{Int}(\text{operand2, e, esize}, unsigned); \\
\text{absdiff} & = \text{Abs}(\text{element1-element2})<\text{esize-1:0}>, \text{pixel} = \text{element1-element2}\text{ bits(esize-1:0)}; \\
\text{Elem[result, e, esize]} & = \text{Elem[result, e, esize]} + \text{absdiff}; \\
\text{V[d]} & = \text{result};
\end{align*}
\]
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.
SABAL, SABAL2

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.

<table>
<thead>
<tr>
<th></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>Rd</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<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>
</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>
</tr>
</tbody>
</table>

Three registers, not all the same type

SABAL(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 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
CheckFPAdvSIMEnabled64();
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.
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 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.

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;
type esize = 8 << UInt(size);
type datasize = if Q == '1' then 128 else 64;
type 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 CPACR_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;
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>; (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.

(old)  htmldiff from-  (new)
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  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 0  | 1  | 1  | 0  | size| 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | Rn | Rd |
| U   | op |

Vector

SADALP <Vd>.<Ta>, <Vn>.<Tb>

declares

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);
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
CheckFPAdvSIMEnabled64();
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.
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.

```
U
0 1 0 0 0 0 1 0 1 0
Rn
Rd

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.

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

```c
MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

if 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;
  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' then
        if CRm<3:2> == '00' && opc == '11' then
          op = MemBarrierOp_SB;
        else
          types = MBReqTypes_All;
          domain = MBReqDomain_FullSystem;
        end;
      end;
    end;
end;
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' then
      if CRm<3:2> == '00' && opc == '11' then
        op = MemBarrierOp_SB;
      else
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;
      end;
    end;
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;
end;
```

case op of
  when MemBarrierOp_DSBDATA_SynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier();
  when MemBarrierOp_SSBSpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_SSSBSpeculativeSynchronizationBarrierToPA();
  when MemBarrierOp_SBSpeculationBarrier();
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.

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</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>Rn</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>
</tr>
</tbody>
</table>

32-bit (sf == 0)

SBC <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SBC <Xd>, <Xn>, < Xm>

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

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

operand2 = NOT(operand2);
if sub_op then
    operand2 = NOT(operand2);

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

<table>
<thead>
<tr>
<th>sf</th>
<th>1</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>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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

SBCS <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SBCS <Xd>, <Xn>, < Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datalength = if sf == '1' then 64 else 32;
integer datalength = if sf == '1' then 64 else 32;
boolean substype = (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>NGCS</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

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

operand2 = NOT(operand2);
if substype then
    operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);
PSTATE.<N,Z,C,V> = nzcv;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.
SBFM

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 \((\text{regsize}-<immr>)\) of the destination register, where \(\text{regsize}\) is the destination register size of 32 or 64 bits.

In both cases the destination bits below 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 \texttt{ASR (immediate)}, \texttt{SBFIZ}, \texttt{SBFX}, \texttt{SXTB}, \texttt{SXTH}, and \texttt{SXTW}.

\[
\begin{array}{cccccccccccccccc}
\text{sf} & | & 0 & 0 & 1 & 0 & 0 & 1 & 0 & N & | & \text{immr} & | & \text{imms} & | & \text{Rn} & | & \text{Rd}
\end{array}
\]

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

\texttt{SBFM <Wd>, <Wn>, #<immr>, #<imms>}

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

\texttt{SBFM <Xd>, <Xn>, #<immr>, #<imms>}

\[
\begin{align*}
\text{integer } & d = \text{UInt}(\text{Rd}); \\
\text{integer } & n = \text{UInt}(\text{Rn}); \\
\text{integer } & \text{datasize} = \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \\
\text{boolean } & \text{inzero}; \\
\text{boolean } & \text{extend}; \\
\text{integer } & R; \\
\text{integer } & S; \\
\text{bits}(&\text{datasize}) & \text{wmask}; \\
\text{bits}(&\text{datasize}) & \text{tmask}; \\
\text{case } & \text{opc of} \\
\text{when } & '00' \text{ inzero } = \text{TRUE}; \text{ extend } = \text{TRUE}; & // \text{SBFM} \\
\text{when } & '01' \text{ inzero } = \text{FALSE}; \text{ extend } = \text{FALSE}; & // \text{BFM} \\
\text{when } & '10' \text{ inzero } = \text{TRUE}; \text{ extend } = \text{FALSE}; & // \text{UBFM} \\
\text{when } & '11' \text{ UNDEFINED}; \\
\end{align*}
\]

\[
\begin{align*}
\text{if } sf == '1' \&\& N != '1' & \text{ then UNDEFINED}; \\
\text{if } sf == '0' \&\& (N != '0' \mid\mid \text{immr}<5> != '0' \mid\mid \text{imms}<5> != '0') & \text{ then UNDEFINED}; \\
R & = \text{UInt}(\text{immr}); \\
S & = \text{UInt}(\text{imms}); \\
(\text{wmask}, \text{tmask}) & = \text{DecodeBitMasks}(N, \text{imms}, \text{immr}, \text{FALSE});
\end{align*}
\]

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 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.

\(<imms>\) For the 64-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 (immediate)</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>SBFZ</td>
<td></td>
<td><strong>UInt</strong> (imms) &lt; <strong>UInt</strong> (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**

```
bits(datasize) src = bits(datasize) dst = if inzero then zeros() else X[d];
bias(datasize) src = X[n];
// perform bitfield move on low bits
bits(datasize) bot = bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);
// determine extension bits (sign, zero or dest register)
bits(datasize) top = 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.
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 \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 Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.
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;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

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);

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);

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);

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);

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);

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);

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);

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);

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);

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);

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);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bots(intsize) intval;

intval = case op of
   when FPConvOp_CVT_FtoI
       fltval = X[n];
       intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
       X[d] = intval;
   when FPConvOp_CVT_ItoF
       intval = X[n];
       fltval = FixedToFP(intval, fracbits, FALSE, FPCR, rounding);
   V[d] = fltval;

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

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>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>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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- sf: Sign bit
- type: Operation type
- Rn: Source register
- Rd: Destination register
- rmode: Rounding mode
- opcode: Instruction code
32-bit to half-precision (sf == 0 && type == 11) (ARMv8.2)
SCVT< Hd>, <Wn>

32-bit to single-precision (sf == 0 && type == 00)
SCVT< Sd>, <Wn>

32-bit to double-precision (sf == 0 && type == 01)
SCVT< Dd>, <Wn>

64-bit to half-precision (sf == 1 && type == 11) (ARMv8.2)
SCVT< Hd>, <Xn>

64-bit to single-precision (sf == 1 && type == 00)
SCVT< Sd>, <Xn>

64-bit to double-precision (sf == 1 && type == 01)
SCVT< 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'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
      rounding = 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(FPCR);
        otherwise
          UNDEFINED;
      otherwise
        UNDEFINED;
    end
  when '01'
    if intsize != 64 else UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
    if opcode<0> == '1' then FPConvOp_CVT_FtoI else FPConvOp_CVT_ItoF;
    part = 0;
    if opcode<0> == '1' then FPConvOp_CVT_FtoI else FPConvOp_CVT_ItoF;
    part = 1;
    if opcode<0> == '1' then FPConvOp_CVT_FtoI else FPConvOp_CVT_ItoF;
    part = 0;
  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.
Operation

```c
CheckFPAdvSIMEnabled64();

bits(FLTSIZE) fltval;
bits(INTSIZE) intval;

intval = case op of
  when FPConvOp_CVT_FtoI
      fltval = V[n];
      intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
      X[n];
      fltval = d = intval;
  when FPConvOp_CVT_ItoF
      intval = X[n];
      fltval = FixedToFP(intval, 0, FALSE, FPCR, rounding); (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<INTSIZE-1:0>;
      Vpart[d, part] = fltval;
  when FPConvOp_CVT_FtoI_JS
      fltval = V[n];
      intval = FPtoFixedJS(fltval, FPCR, TRUE);
      V[d] = ZeroExtend[d] = fltval; (intval<31:0>, 64);
```
SDIV

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;
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];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
  result = 0;
else
  result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | size | L  | M  | Rm | 1  | 1  | 1  | 0  | H  | 0  | Rn | Rd |

Vector

SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]

```plaintext
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
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>&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 "M:Rm" fields.

<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]);
        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;
```

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1 | 1 | 0 | size | 0 | Rm | 1 | 0 | 0 | 1 | 0 | 1 | Rn | 1 | 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');
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.
Operation

```
CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = \texttt{V}[n];
bits(datasize) operand2 = \texttt{V}[m];
bits(datasize) result;

result = \texttt{V}[d];
for e = 0 to elements-1
  integer res = 0;
  integer element1, element2;
  for i = 0 to 3
    if signed then
      element1 = \texttt{SInt(Elem[operand1, 4*e+i, esize DIV 4])};
    else
      element1 = \texttt{UInt(Elem[operand1, 4*e+i, esize DIV 4])};
    endif
    element2 = \texttt{SInt(Elem[operand2, 4*e+i, esize DIV 4])};
    else
      element2 = \texttt{UInt(Elem[operand2, 4*e+i, esize DIV 4])};
    endif
    res = res + element1 * element2;
    \texttt{Elem[result, e, esize] = Elem[result, e, esize] + res;}
  \texttt{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)

<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>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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **sf**

**SETF8** *(sz == 0)*

```assembly
SETF8 <Wn>
```

**SETF16** *(sz == 1)*

```assembly
SETF16 <Wn>
```

```assembly
if !HaveFlagManipulateExt() 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**

```assembly
bits(32) tmpreg = X[n];
PSTATE.N = tmpreg<msb>;
PSTATE.Z = if (tmpreg<msb:0> == Zeros(msb + 1)) then '1' else '0';
PSTATE.V = tmpreg<msb+1> EOR tmpreg<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.

```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, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0' op = SystemHintOp_BT;
    BTypeCompatible = BTypeCompatible_BT(op2<2:1>);
  otherwise EndOfInstruction// Empty(); // Instruction executes as
```

System
case op of
    when SystemHintOp_YIELD() :
        when SystemHintOp_WFE
            if !System::IsEventRegisterSet() then
                ClearEventRegister();
            else
                if PSTATE.EL = EL3 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 !System::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();
                if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
                    AArch64.vESBOperation();
                TakeUnmaskedErrorInterrupts();
            when SystemHintOp_PSB
                ProfilingSynchronizationBarrier();
            when SystemHintOp_TSB
                TraceSynchronizationBarrier();
            when SystemHintOp_CSDB
                ConsumptionOfSpeculativeDataBarrier();
            when SystemHintOp_BTI
                BTypeNext = '00';
            otherwise // do nothing

SEVL

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.

System

```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, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
      op = SystemHintOp_BTI;
      BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction(); // Empty instruction executes as
```
case op of
  when SystemHintOp_YIELD();
    if EventRegisterSet() 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(EL2) && PSTATE_EL == EL2 then
          // Check for traps described by the Secure Monitor.
          AArch64.CheckForWFxTrap(EL2, TRUE);
          WaitForEvent();
    when SystemHintOp_WFE
      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(EL2) && PSTATE_EL == EL2 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFxTrap(EL2, FALSE);
            WaitForInterrupt();
    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(EL2) && PSTATE_EL == EL2 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFxTrap(EL2, FALSE);
            WaitForInterrupt();
        when SystemHintOp_SEVSendEvent();
        when SystemHintOp_SEVLSendEventLocal();
        when SystemHintOp_ESBSynchronizeError();
          AArch64.ESBOperation();
          if El2Enabled() || PSTATE_EL IN (EL0, EL1) then
            AArch64.vESBOperation();
            TakeUnmaskedErrorInterrupts();
        when SystemHintOp_CSBSynchronizingSynchronizationBarrier();
        when SystemHintOp_TSB.
          TraceSynchronizationBarrier();
        when SystemHintOp_CSSBConsumptionOfSpeculativeDataBarrier();
        when SystemHintOp_BT(); #TypeNext = '00';
        otherwise // do nothing

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA1C

SHA1 hash update (choose).

<table>
<thead>
<tr>
<th></th>
<th>1</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>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>
</table>

Advanced SIMD

SHA1C <Qd>, <Sn>, <Vm>.4S

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 = SHA1Choose(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); // 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.
SHA1M

SHA1 hash update (majority).

<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>Rm</td>
<td>0</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>
</tr>
</tbody>
</table>

Advanced SIMD

SHA1M <Qd>, <Sn>, <Vm>.4S

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 = SHAmajority(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.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  |

Advanced SIMD

SHA1P <Qd>, <Sn>, <Vm>.4S

```
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 = 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);  // 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  |  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 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1   | 0   | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  |

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 = 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.
SHA256H2

SHA256 hash update (part 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  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | Rm | 0  | 1  | 0  | 0  | Rn | Rd |

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;
result = if part1 then
   result = SHA256hash(V[n],V[d], V[m], FALSE);4[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.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA256H

SHA256 hash update (part 1).

Advanced SIMD

SHA256H <Qd>, <Qn>, <Vm>.4S

```plaintext
type integer d = UInt(Rd);
type integer n = UInt(Rn);
type 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;
result = if part1 then
    result = SHA256hash(V[d], V[n], V[m], TRUE);[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.
SHA256SU0

SHA256 schedule update 0.

<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>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>1</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>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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(128) T = operand2<31:0> : operand1<127:32>;
bits(32) elt;
for e = 0 to 3
  elt = E|em[T, e, 32];
  elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
  E|lem[result, e, 32] = elt + E|em[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.

<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>0</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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
bits(128) T0 = operand3<31:0>:operand2<127:32>;
bits(128) T0 = operand3<31:0>:operand2<127:32>;
bits(64) T1;
bits(32) elt;
T1 = operand3<127:64>;
for e = 0 to 1
   elt = Elem[T1, e, 32];
   elt = XOR(elt, 17) XOR XOR(elt, 19) XOR LSR(elt, 10);
   elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
   Elem[result, e, 32] = elt;
T1 = result<63:0>;
for e = 2 to 3
   elt = Elem[T1, e-2, 32];
   elt = XOR(elt, 17) XOR XOR(elt, 19) XOR LSR(elt, 10);
   elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
   Elem[result, e, 32] = elt;
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.
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

**SHA512H2**

<Qd>, <Qn>, <Vm>.2D

```c
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**

```c
ARArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
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>, 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>, 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.
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)

```
<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>
</tr>
</thead>
<tbody>
<tr>
<td>1</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Rm  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
Rn  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
Rd  |
```

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();

Vtmp = (V[n] AND X<63:0>) EOR (NOT(V[n]) AND X<127:64>); 
Vtmp = (Vtmp + MSigma1 + W<127:64>);

Vtmp = (V[n] AND X<63:0>) EOR (NOT(V[n]) AND X<127:64>); 
Vtmp = (Vtmp + MSigma1 + W<127:64>);

MSigma1 = ROR(Y<127:64>, 14) EOR ROR(Y<127:64>, 18) EOR ROR(Y<127:64>, 41);
Vtmp = (V[n] AND X<63:0>) EOR (NOT(V[n]) AND X<127:64>); 
Vtmp = (Vtmp + MSigma1 + W<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.
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  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  |

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];

tmp1 = ROBX<X<127:64>>, 19) XOR ROBX<X<63:0>>, 61) XOR ('000000':X<127:70>)
\(\text{OR}(\text{X}[127:64], 19) \text{EOR} \text{X}[63:0], 61) \text{EOR} ('000000':\text{X}[127:70])\)
Vtmp<127:64> = W<127:64> + sig1 + Y<127:64>;
sig1 = ROBX<X<63:0>>, 19) XOR ROBX<X<63:0>>, 61) XOR ('000000':X<63:6>)
\(\text{OR}(\text{X}[63:0], 19) \text{EOR} \text{X}[63:0], 61) \text{EOR} ('000000':\text{X}[63:6])\)
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.
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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  |

Advanced SIMD

SM3PARTW1 <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;

result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):Vm<127:96>,15):ROL(Vm<95:64>, 15):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 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.
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)

<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>0</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>1</td>
<td>0</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>0</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>
</tr>
</tbody>
</table>

Advanced SIMD

SM3PARTW2 <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;
bits(128) tmp;
bits(32) temp2;
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>;
temp2 = ROL(tmp<31:0>, 15);
tmp2 = temp2 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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 | 0  | Ra | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

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];
V<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>, 7);  
(Wm<127:96>, 12) = Vm<127:96> + Va<127:96>  
V<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  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 1  | 0  | imm2| 0  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Advanced SIMD

SM3TT1A <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

```c
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>);

result<31:0> = Vd<63:32>;
result<31:0> = 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.
SM3TT1B

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)

<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>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>1</td>
<td>0</td>
<td>imm2</td>
<td>0</td>
<td>1</td>
<td>Rm</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>

SM3TT1B <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

```c
AArch64.CheckFPAAdvSIMDEnabled();

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<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.

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | Rm | 1  | 0  | imm2 | 1  | 0  | Rn |    |    |    |   |    |    |    |  Rd |

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];
bets(128) Vn = V[n];
bets(128) Vd = V[d];
bets(32) Wj;
bets(128) result;
bits(128) result;
bets(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);(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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 | 1  | 1  | Rn | | | | | | | | | | | | | | | | | | | |

Advanced SIMD


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, 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.

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  |

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(32) intval;
bits(8) sboxout;
bits(128) roundresult;
bits(32) roundkey;

roundresult = V[d];
for index = 0 to 3
    roundkey = Elem[Vn, index, 32];
    [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, i, 8];

intval = intval EOR ROL(intval, 2) EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 10) EOR ROL(intval, 24);
[intval,24];

intval = intval EOR ROL(intval, 2) EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 10) EOR ROL(intval, 24);
[intval,24];

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.
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)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<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>Rn</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>Rm</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

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.
SMADDL

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.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 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   | 0   | 0   | 0   | 1   | Rm  | 0   | Ra  | Rn  | Rd  | U   | 00  |

64-bit

SMADDL <Xd>, <Wn>, <Wm>, <Xa>

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>SMULL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

```
bits(32) operand1 = bits(datasize) operand1 = X[n];
bits(32) operand2 = bits(datasize) operand2 = X[m];
bits(64) operand3 = bits(destsize) operand3 = X[a];

integer result;

result = if sub_op then
    result = Int(operand3, FALSE) + ((operand3, unsigned) - Int(operand1, FALSE) * (operand1, 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.
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 *HCR_EL2.TSC* and *SCR_EL3.SMD* are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in *ESR_ELx*, using the EC value 0x17, that is taken to EL3.

If the value of *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 *SCR_EL3.SMD*. For more information, see *Traps to EL2 of Non-secure EL1 execution of SMC instructions*.

If the value of *HCR_EL2.TSC* is 0 and the value of *SCR_EL3.SMD* is 1, the SMC instruction is **UNDEFINED**.

- System

  SMC #<imm>

  ```
  // Empty. 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.CheckForSMCUndefOrTrap(imm16);
  (imm);

  if SCR_EL3.SMD == '1' then
    // SMC disabled
    AArch64.UndefinedFault();
  else
    AArch64.CallSecureMonitor(imm16);(imm);
  ```
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.

```

Vector

SMLAL[2] <Vd>, <Ta>, <Vn>,<Tb>, <Vm>.<Ts>[<index>]

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
\text{case size of}
  \text{when '01' index = UInt(H:L:M); Rmhi = '0';}
  \text{when '10' index = UInt(H:L); Rmhi = M;}
\text{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

\text{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{tabular}{c|c}
Q & 2 \\
\hline
0 & [absent] \\
1 & [present] \\
\end{tabular}

<Vd> \text{Is the name of the SIMD&FP destination register, encoded in the “Rd” field.}
<Ta> \text{Is an arrangement specifier, encoded in “size”:

\begin{tabular}{c|c}
size & <Ta> \\
\hline
00 & RESERVED \\
01 & 4S \\
10 & 2D \\
11 & RESERVED \\
\end{tabular}

<Vn> \text{Is the name of the first SIMD&FP source register, encoded in the “Rn” field.}
<Tb> \text{Is an arrangement specifier, encoded in “size:Q”:

\begin{tabular}{c|c}
size & <Ta> \\
\hline
\end{tabular}

\text{SMLAL, SMLAL2 (by element)}

Page 783
```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Th&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.
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 \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{table}
\centering
\begin{tabular}{c|c|c|c|c|c}
\hline
31 & 30 & 29 & 28 & 27 & 26 \hline
25 & 24 & 23 & 22 & 21 & 20 \hline
19 & 18 & 17 & 16 & 15 & 14 \hline
13 & 12 & 11 & 10 & 9 & 8 \hline
7 & 6 & 5 & 4 & 3 & 2 \hline
1 & 0 & 0 & 0 & 0 & 0 \hline
\end{tabular}
\caption{Addressing modes for SMLAL, SMLAL2 (vector)}
\end{table}

Three registers, not all the same type

\begin{verbatim}
SMLAL(2), <Vn>, <Ta>, <Vm>, <Tb>
\end{verbatim}

\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 = 64;
integer part = Uint(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
\end{verbatim}

\subsubsection{Assembler Symbols}

\begin{table}
\centering
\begin{tabular}{c|c}
\hline
Q & 2 \\
\hline
0 & [absent] \\
1 & [present] \\
\end{tabular}
\caption{Asm Q symbol}
\end{table}

\begin{verbatim}<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\end{verbatim}

\begin{verbatim}<Ta> Is an arrangement specifier, encoded in “size”:
\end{verbatim}

\begin{verbatim}
\begin{tabular}{c|c}
\hline
size & <Ta> \\
\hline
00 & 8H \\
01 & 4S \\
10 & 2D \\
11 & RESERVED \\
\end{tabular}
\caption{Asm size (Q symbol)}
\end{table}

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

\begin{verbatim}<Tb> Is an arrangement specifier, encoded in “size:Q”:
\end{verbatim}

\begin{verbatim}
\begin{tabular}{c|c|c}
\hline
size & Q & <Tb> \\
\hline
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & RESERVED \\
\end{tabular}
\caption{Asm size:Q symbol}
\end{table}

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

```
<table>
<thead>
<tr>
<th></th>
<th>Q</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>O2</th>
<th>H</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

Vector

SMLSL[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

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>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”:
<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>00</th>
<th>RESERVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0:Rm</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
<td></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>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();
bias(datasize) operand1 = Vpart[n, part];
bias(idxdsize) operand2 = V[m];
bias(2*datasize) operand3 = V[d];
bias(2*datasize) result;
integer element1;
integer element2;
bias(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>>;
    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.
SMLSL, SMLSL2 (vector)

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  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U   | Q  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm | 1  | 0  | 1  | 0  | 0  | Rn | Rd |

Three registers, not all the same type

SMLSL[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 sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols


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”:

<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.
SMSUBL

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  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |

64-bit

SMSUBL <Xd>, <Wn>, <Wm>, <Xa>

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 minuend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMNEGL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

```
integer result;
result = if sub_op then
    result = Int(operand3, FALSE) - (Int(operand3, unsigned) - Int(operand1, FALSE) * Int(operand1, unsigned));
else
    result = Int(operand1, unsigned) + Int(operand2, unsigned) * Int(operand3, 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.

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 datasize = 64;
integer destsize = 64;
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(64) operand1 = bits(datasize) operand1 = X[n];
bits(64) operand2 = bits(datasize) operand2 = X[m];
integer result;
result = Int(operand1, FALSE) *{operand1, unsigned} * Int(operand2, FALSE);{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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | L  | M  | Rm | 1  | 0  | 1  | 0  | H  | 0  | Rn |  | Rd |

Vector

\[
\text{SMULL} \{2\} \langle \text{Vd}\rangle, \langle \text{Ta}\rangle, \langle \text{Vn}\rangle, \langle \text{Tb}\rangle, \langle \text{Vm}\rangle, \langle \text{Ts}\rangle[\langle \text{index}\rangle]
\]

```plaintext
inginteger idxdsiz = if H == '1' then 128 else 64;
inginteger index;
bing 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);
inginteger n = UInt(Rn);
inginteger m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
inginteger datasize = 64;
inginteger part = UInt(Q);
inginteger elements = datasize DIV esize;
ingboolean 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>

\(<\text{Vd}\rangle\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{Ta}\rangle\) 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>

\(<\text{Vn}\rangle\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Tb}\rangle\) 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>

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) 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.

Three registers, not all the same type

SMULL(2) <Vd>,<Ta>, <Vn>,<Tb>, <Vm>.

\[
\begin{align*}
\text{integer } d & = \text{UInt}(Rd); \\
\text{integer } n & = \text{UInt}(Rn); \\
\text{integer } m & = \text{UInt}(Rm); \\
\text{if } \text{size} == '11' \text{ then UNDEFINED; } \\
\text{integer } \text{esize} & = 8 \text{ } \ll \text{UInt(size)}; \\
\text{integer } \text{datasize} & = 64; \\
\text{integer } \text{part} & = \text{UInt}(Q); \\
\text{integer } \text{elements} & = \text{datasize} \text{ } \text{DIV} \text{ } \text{esize}; \\
\text{boolean } \text{unsigned} & = (U == '1');
\end{align*}
\]

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":

\[
\begin{array}{c|c}
Q & 2 \\
0 & \text{[absent]} \\
1 & \text{[present]} \\
\end{array}
\]

<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.

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

\[
\begin{array}{ccccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 1 & \text{size} & L & M & Rm & 0 & 0 & 1 & 1 & H & 0 & Rn & Rd & o2
\end{array}
\]

#### Scalar

\[
\text{SQDMLAL } \langle \text{Va} \rangle \langle d \rangle, \langle \text{Vb} \rangle \langle n \rangle, \langle \text{Vm} \rangle \langle \text{Ts} \rangle[\langle \text{index} \rangle]
\]

```
integer idxdsze = 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

\[
\begin{array}{ccccccccc}
0 & Q & 0 & 0 & 1 & 1 & 1 & \text{size} & L & M & Rm & 0 & 0 & 1 & 1 & H & 0 & Rn & Rd & o2
\end{array}
\]
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>O: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(idxdsiz) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
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;
```

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, CPTCR_EL2, and CPTCR_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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>
<tr>
<td></td>
<td></td>
<td></td>
<td></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>
</tr>
</tbody>
</table>

**Scalar**

\[
\text{SQDMLAL} \langle V_a\rangle<d>, \langle V_b\rangle<n>, \langle V_b\rangle<m>
\]

- integer \(d\) = \(\text{UInt}(Rd)\);
- integer \(n\) = \(\text{UInt}(Rn)\);
- integer \(m\) = \(\text{UInt}(Rm)\);

```plaintext
if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << \(\text{UInt}(size)\);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '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>1</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>0</td>
<td>1</td>
<td>0</td>
<td>0</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>
<tr>
<td></td>
<td></td>
<td></td>
<td></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>
</tr>
</tbody>
</table>

**Vector**

\[
\text{SQDMLAL(2)} \langle V_d\rangle.<T_a>, \langle V_n\rangle.<T_b>, \langle V_m\rangle.<T_b>
\]

- integer \(d\) = \(\text{UInt}(Rd)\);
- integer \(n\) = \(\text{UInt}(Rn)\);
- integer \(m\) = \(\text{UInt}(Rm)\);

```plaintext
if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << \(\text{UInt}(size)\);
integer datasize = 64;
integer part = \(\text{UInt}(Q)\);
integer elements = datasize \(\text{DIV}\) 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”:

<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>Tb</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*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);
    (2 * element1 * element2, 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: 2018-09-13T13:25

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

```
<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>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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</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>
```

Scalar

```
int idxdsize = if H == '1' then 128 else 64;
int 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;
int d = UInt(Rd);
int n = UInt(Rn);
int m = UInt(Rmhi:Rm);
int esize = 8 << UInt(size);
int datasize = esize;
int elements = 1;
int part = 0;
bool sub_op = (o2 == '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>1</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>H</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '0' 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(idxsSize) operand2 = V[m];
bits(2*dataSize) operand3 = V[d];
bits(2*dataSize) result;
integer element1;
integer element2;
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);
    (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 (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 0 | size | 1 | Rm | 1 0 1 1 0 0 | Rn | 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 0 1 0 1 1 1 0 | size | 1 | Rm | 1 0 1 1 0 0 | Rn | Rd
```

### Scalar

```
SQDMLSL <Va><d>, <Vb><n>, <Vb><m>

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

```
SQDMLSL(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

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”:

<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 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];
bright(datasize) operand2 = Vpart[m, part];
bright(2*datasize) operand3 = V[d];
bright(2*datasize) result;
integer element1;
integer element2;
bright(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;
```

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 $FPSR.QC$ is set.

The SQDMULL instruction extracts the first source vector from the lower half of the first source register, while the SQDMULL2 instruction extracts the first source vector 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 | 1 | 0 | 1 | 1 | H | 0 | Rn | Rd |

### Scalar

```
SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

```plaintext
integer idxdsig = 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;
```

### 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 | size | L | M | Rm | 1 | 0 | 1 | 1 | H | 0 | Rn | Rd |
```
Vector

\textbf{SQDMULL(2)} \textlangle Vd\rangle, \textlangle Vn\rangle, \textlangle Tb\rangle, \textlangle Vm\rangle, \textlangle Ts\rangle[\textlangle index\rangle]

\begin{verbatim}
integer idxdsiz = if \textbackslash H == \textquoteleft1\textbackslash then 128 else 64;
integer index;
bit Rmhi;
case size of
  when \textquoteleft01\textbackslash index = UInt(H:L:M); Rmhi = \textquoteleft0\textbackslash;
  when \textquoteleft10\textbackslash 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;
\end{verbatim}

\textbf{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 \textquoteleft Q\textbackslash:

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
0 & \text{[absent]} \\
1 & \text{[present]} \\
\hline
\end{tabular}
\end{table}

\textlangle Vd\rangle Is the name of the SIMD\&FP destination register, encoded in the \textbackslash Rd\textbackslash field.

\textlangle Ta\rangle Is an arrangement specifier, encoded in \textquoteleft size\textbackslash:

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
00 & RESERVED \\
01 & 4S \\
10 & 2D \\
11 & RESERVED \\
\hline
\end{tabular}
\end{table}

\textlangle Vn\rangle Is the name of the first SIMD\&FP source register, encoded in the \textbackslash Rn\textbackslash field.

\textlangle Tb\rangle Is an arrangement specifier, encoded in \textquoteleft size:Q\textbackslash:

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
00 & x \text{RESERVED} \\
01 & 0 \text{4H} \\
01 & 1 \text{8H} \\
10 & 0 \text{2S} \\
10 & 1 \text{4S} \\
11 & x \text{RESERVED} \\
\hline
\end{tabular}
\end{table}

\textlangle Va\rangle Is the destination width specifier, encoded in \textquoteleft size\textbackslash:

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
00 & RESERVED \\
01 & S \\
10 & D \\
11 & RESERVED \\
\hline
\end{tabular}
\end{table}

\textlangle d\rangle Is the number of the SIMD\&FP destination register, encoded in the \textbackslash Rd\textbackslash field.

\textlangle Vb\rangle Is the source width specifier, encoded in \textquoteleft size\textbackslash:

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
00 & RESERVED \\
01 & H \\
10 & S \\
11 & RESERVED \\
\hline
\end{tabular}
\end{table}

\textlangle n\rangle Is the number of the first SIMD\&FP source register, encoded in the \textbackslash Rn\textbackslash field.

\textlangle Vm\rangle Is the name of the second SIMD\&FP source register, encoded in \textquoteleft size:M:Rm\textbackslash: 
<Vm>

Restricted to V0-V15 when element size <Ts> is H.

<index>

Is the element index, encoded in “size:L:H:M”:

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(idxsizesize) 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;
```
**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 $FP_{SR}.QC$ 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 1  | 0  | 1  | 0  | 0  | Rn | 1  | 1  | 0  | 1  | 0  | 0  | Rd | 1  | 1  |

### Scalar

```
SQDMULL <Va><d>, <Vb><n>, <Vb><m>
```

- 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;

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

### Vector

```
SQDMULL<2> <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

- 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;

### 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.
Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>Ta</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>

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>Tb</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>4H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>2S</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>
<th>Va</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>

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>
<th>Vb</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 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.

**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) 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;
```

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

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | size | 1 | Rm | 0  | 0  | 0  | 1  | 0  | 1  | Rn | Rd |

**Three registers of the same type**

SRHADD `<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**

<br>

**Operation**

```c
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;
```
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.

```assembly
31 30 28 27 26 25 24 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 1 1 0 0 0 1 1 0 0 1 1 1 1
CRm opc
```

System

```assembly
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;
    elseif CRm<3:2> == '00' && opc == '00' then
      op = MemBarrierOp_SSBB;
    elseif HaveSBExt() && CRm<3:2> == '00' && opc == '11' then
      op = MemBarrierOp_SB;
    else
      types = MBReqTypes_All;
      domain = MBReqDomain_FullSystem;// Empty.
  end;
```
case op of
    when MemBarrierOp_DSBB : DataSynchronizationBarrier (domain, types);
    when MemBarrierOp_DMB : DataMemoryBarrier (domain, types);
    when MemBarrierOp_ISBB : InstructionSynchronizationBarrier ();
    when MemBarrierOp_SSBB : SpeculativeSynchronizationBarrierToVA ();
    when MemBarrierOp_PSSBB : SpeculativeSynchronizationBarrierToPA ();
    when MemBarrierOp_SB : SpeculationBarrier ();
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

| 31 30 29 28 27 26 25 24 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 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 1 | x | size | Rn | Rt |

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

| 31 30 29 28 27 26 25 24 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 | 0 | 0 | 0 | Rm | x | x | 1 | x | size | Rn | Rt |

L | opcode

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
One register, immediate offset (Rm == 1111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 1111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 1111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 1111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 1111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 1111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 1111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 1111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```c
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":

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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”:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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”:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, 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.
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   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | x   | x   | 0   | S   | size | Rn  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

L R opcode

8-bit (opcode == 000)

| ST1 { <Vt>.B }[<index>], [<Xn|SP>] |

16-bit (opcode == 010 && size == x0)

| ST1 { <Vt>.H }[<index>], [<Xn|SP>] |

32-bit (opcode == 100 && size == 00)

| ST1 { <Vt>.S }[<index>], [<Xn|SP>] |

64-bit (opcode == 100 && S == 0 && size == 01)

| ST1 { <Vt>.D }[<index>], [<Xn|SP>] |

```plaintext
t = UInt(Rt);
n = UInt(Rn);
m = integer UNKNOWN;
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   | 0   | 1   | 1   | 0   | 0   | Rm  | x   | x   | 0   | S   | size | Rn  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

L R opcode
8-bit, immediate offset (Rm == 11111 && opcode == 000)

\[
\text{ST1} \{ <\text{Vt}.B> \}[<\text{index}>], [<\text{Xn}|\text{SP}>], \#1
\]

8-bit, register offset (Rm != 11111 && opcode == 000)

\[
\text{ST1} \{ <\text{Vt}.B> \}[<\text{index}>], [<\text{Xn}|\text{SP}>], <\text{Xm}>
\]

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

\[
\text{ST1} \{ <\text{Vt}.H> \}[<\text{index}>], [<\text{Xn}|\text{SP}>], \#2
\]

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

\[
\text{ST1} \{ <\text{Vt}.H> \}[<\text{index}>], [<\text{Xn}|\text{SP}>], <\text{Xm}>
\]

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

\[
\text{ST1} \{ <\text{Vt}.S> \}[<\text{index}>], [<\text{Xn}|\text{SP}>], \#4
\]

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

\[
\text{ST1} \{ <\text{Vt}.S> \}[<\text{index}>], [<\text{Xn}|\text{SP}>], <\text{Xm}>
\]

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

\[
\text{ST1} \{ <\text{Vt}.D> \}[<\text{index}>], [<\text{Xn}|\text{SP}>], \#8
\]

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

\[
\text{ST1} \{ <\text{Vt}.D> \}[<\text{index}>], [<\text{Xn}|\text{SP}>], <\text{Xm}>
\]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;

Assembler Symbols

\(<\text{Vt}>\) Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<\text{index}>\) 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".
\(<\text{Xn}|\text{SP}>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\(<\text{Xm}>\) Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
shared decode

integer scale = \texttt{UInt}(\texttt{opcode}<2:1>);
integer selem = \texttt{UInt}(\texttt{opcode}<0>:R) + 1;
boolean replicate = \texttt{FALSE};
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = \texttt{UInt}(\texttt{size});
    replicate = TRUE;
  when 0
    index = \texttt{UInt}(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = \texttt{UInt}(Q:S: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;
  
\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;
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, 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, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, 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

```
<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>0</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>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>
<td></td>
</tr>
</tbody>
</table>
```

L

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

No offset

```
ST2 { <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>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>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>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

L

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rm</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

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

- `<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>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;
```

ST2 (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, address+offs, ebytes, AccType_VEC];
        V[tt] = rval;
      else // memop == MemOp_STORE
        Mem[address+offs, ebytes, 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 \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**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | x  | x  | 0  | S  | size | Rn | Rt |
| L  | R  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

8-bit (opcode == 000)

\[
\text{ST2} \{ <Vt>.B, <Vt2>.B \}[<index>], [<Xn|SP>] 
\]

16-bit (opcode == 010 && size == x0)

\[
\text{ST2} \{ <Vt>.H, <Vt2>.H \}[<index>], [<Xn|SP>] 
\]

32-bit (opcode == 100 && size == 00)

\[
\text{ST2} \{ <Vt>.S, <Vt2>.S \}[<index>], [<Xn|SP>] 
\]

64-bit (opcode == 100 && S == 0 && size == 01)

\[
\text{ST2} \{ <Vt>.D, <Vt2>.D \}[<index>], [<Xn|SP>] 
\]

integer \( t = \text{UInt}(\text{Rt}) \);
integer \( n = \text{UInt}(\text{Rn}) \);
integer \( m = \text{integer \text{UNKNOWN}} \);
boolean \( \text{wback} = \text{FALSE} \);

**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  | 1  | 0  | 1  | 0  | 0  | 0  | x  | x  | 0  | S  | size | Rn | Rt |
| L  | R  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

integer \( t = \text{ UInt}(\text{Rt}) \);
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.
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, address + offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
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, address + offs, ebytes, AccType_VEC];
        V[t] = rval;
else
    // memop == MemOp_STORE
    // extract from one lane of 128-bit register
    Mem[address+offs, ebytes, 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)

<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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>imm9</td>
<td>0</td>
<td>1</td>
<td>Xn</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>
</tr>
</tbody>
</table>

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)

<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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>imm9</td>
<td>1</td>
<td>1</td>
<td>Xn</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>
</tr>
</tbody>
</table>

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)

<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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>imm9</td>
<td>1</td>
<td>0</td>
<td>Xn</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>
</tr>
</tbody>
</table>

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

```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: 2018-09-13T13:25:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 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  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 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

<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>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<V2>
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 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>&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>

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 = 3; // LD/ST3 (3 registers)
when '1000' rpt = 1; selem = 2; // LD/ST1 (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;
```

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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, 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.

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

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

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
8-bit, immediate offset (Rm == 11111 && opcode == 001)

\[ \text{ST3} \{ <Vt>.B, <Vt2>.B, <Vt3>.B \}[<index>], [<Xn|SP>], #3 \]

8-bit, register offset (Rm != 11111 && opcode == 001)

\[ \text{ST3} \{ <Vt>.B, <Vt2>.B, <Vt3>.B \}[<index>], [<Xn|SP>], <Xm> \]

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

\[ \text{ST3} \{ <Vt>.H, <Vt2>.H, <Vt3>.H \}[<index>], [<Xn|SP>], #6 \]

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

\[ \text{ST3} \{ <Vt>.H, <Vt2>.H, <Vt3>.H \}[<index>], [<Xn|SP>], <Xm> \]

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

\[ \text{ST3} \{ <Vt>.S, <Vt2>.S, <Vt3>.S \}[<index>], [<Xn|SP>], #12 \]

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

\[ \text{ST3} \{ <Vt>.S, <Vt2>.S, <Vt3>.S \}[<index>], [<Xn|SP>], <Xm> \]

64-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 0 & & size == 01)

\[ \text{ST3} \{ <Vt>.D, <Vt2>.D, <Vt3>.D \}[<index>], [<Xn|SP>], #24 \]

64-bit, register offset (Rm != 11111 && opcode == 101 && size == 0 & & size == 01)

\[ \text{ST3} \{ <Vt>.D, <Vt2>.D, <Vt3>.D \}[<index>], [<Xn|SP>], <Xm> \]

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.
integer scale = \texttt{UInt}(\texttt{opcode<2:1>});
integer selem = \texttt{UInt}(\texttt{opcode<0>:R}) + 1;
boolean replicate = FALSE;
integer index;

\textbf{case scale of}
\begin{enumerate}
\item when 3
  \begin{enumerate}
  \item if \texttt{L} == '0' \textbf{||} \texttt{S} == '1' then UNDEFINED;
  \item scale = \texttt{UInt}(\texttt{size});
  \item replicate = TRUE;
  \end{enumerate}
\item when 0
  \begin{enumerate}
  \item index = \texttt{UInt}(\texttt{Q:S:size}); \textbf{// B[0-15]}
  \end{enumerate}
\item when 1
  \begin{enumerate}
  \item if \texttt{size<0>} == '1' then UNDEFINED;
  \item index = \texttt{UInt}(\texttt{Q:S:size<1>}); \textbf{// H[0-7]}
  \end{enumerate}
\item when 2
  \begin{enumerate}
  \item if \texttt{size<1>} == '1' then UNDEFINED;
  \item if \texttt{size<0>} == '0' then
    \begin{enumerate}
    \item index = \texttt{UInt}(\texttt{Q:S}); \textbf{// S[0-3]}
    \end{enumerate}
  \item else
    \begin{enumerate}
    \item if \texttt{S} == '1' then UNDEFINED;
    \item index = \texttt{UInt}(\texttt{Q}); \textbf{// D[0-1]}
    \item scale = 3;
    \end{enumerate}
  \end{enumerate}
\end{enumerate}

\texttt{MemOp} memop = if \texttt{L} == '1' then \texttt{MemOp\_LOAD} else \texttt{MemOp\_STORE};
integer datasize = if \texttt{Q} == '1' then 128 else 64;
integer esize = 8 \texttt{<< 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, 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, address + offs, ebytes, AccType_VEC];
                V[t] = rval;
            else if memop == MemOp_STORE then
                // extract from one lane of 128-bit register
                Mem[address+offs, ebytes, 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 0 0 0 | size | Rn | Rt |
```

```
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 0 0 0 0 | Rm | size | Rn | Rt |
```

### Immediate offset (Rm == 11111)

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
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

- `<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>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|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>#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 = 3; // 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;
```

ST4 (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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, 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 \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: \textbf{No offset} and \textbf{Post-index}

\textbf{No offset}

\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
\hline 0 & Q & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & x & x & 1 & S & size & Rn & Rt \\
\hline
L & R & \multicolumn{2}{c}{opcode}
\end{verbatim}

8-bit (opcode == 001)


16-bit (opcode == 011 && size == x0)


32-bit (opcode == 101 && size == 00)


64-bit (opcode == 101 && S == 0 && size == 01)


\begin{verbatim}
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
\end{verbatim}

\textbf{Post-index}

\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
\hline 0 & Q & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & x & x & 1 & S & size & Rn & Rt \\
\hline
L & R & \multicolumn{2}{c}{opcode}
\end{verbatim}
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)


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, 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, address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, 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.
STG

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 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 1
imm9 0 1 Xn (1) (1) (1) (1) (1)
```

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 0 1 1 1 1 0 0 1 0 0 1 1 1 1 0 0 1 0 0 1 1 1 1 0 0 1
imm9 1 1 Xn (1) (1) (1) (1) (1)
```

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 1 0 0 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1
imm9 1 0 Xn (1) (1) (1) (1) (1)
```

STG
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
    CheckSAPAlignment();
    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;
```

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)

<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>L</td>
<td>Rs</td>
<td>o0</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>
</tr>
</tbody>
</table>

size

32-bit (size == 10)

STLLR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STLLR <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 "Rt" field.
Operation

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
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 = case memop of
    when MemOp_STORE
    data = X[t];
    Mem[address, dbytes, [address, dbytes, acctype] = data;
    when
    data = Mem[address, dbytes, acctype],
    x[t] = ZeroExtendAccType_LIMITEDORDERED(MemOp_LOAD = data, 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: 2018-09-13T13:25
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>0</td>
<td>0</td>
<td>1</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>0</td>
<td>(1)</td>
<td>(1)</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>
</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>size</td>
<td>L</td>
<td>Rs</td>
<td>o0</td>
<td>Rt2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

STLLRB <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(8) data;
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 = case memop of
    when MemOp_STORE
        data = X[t];
    Mem[address, dbyte, acctype] = data;
    when
        data = Mem[address, dbyte, acctype];
    Mem[address, dbyte, acctype] = data;

if HaveMTEExt() then
    ZeroExtAndAccType_LIMITEDORDERED[MemOp_LOAD] = 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)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>0</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>L</td>
<td>Rs</td>
<td>o0</td>
<td>Rt2</td>
<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

STLLRH <Wt>, [<Xn|SP>(, #0])

integer n = UInt(Rn);
integer t = UInt(Rt); // ignored by load/store single register
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(16) data;
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 = case memop of
    when MemOp_STORE
        data = X[t];
    Mem[address, 2, address, dbyte, acctype] = data;
    when
        data = Mem[address, dbyte, acctype];
        X[t] = ZeroExtend(AccType_LIMITEDORDERED[MemOp_LOAD] = 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.

<table>
<thead>
<tr>
<th>32-bit (size == 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STLR &lt;Wt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit (size == 11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STLR &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.

**Operation**

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer datasize = elsize;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    MemOp_LOAD =>
        data = X[t];
    MemOp_STORE =>
        data = Mem[address, dbytes, acctype = data];
when
    data = Mem[address, dbytes, acctype = data];
    X[t] = ZeroExtendAccType_ORDERED(MemOp_LOAD) = data; datasize, regsize);
```

Page 860
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  | 0  | 0  | 1  | 0  | 1  | (1) | (1) | (1) | (1) | 1  | (1) | (1) | (1) | (1) | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| size | L | Rs | o0 | Rt2 |

No offset

STLRB <Wt>, [<Xn|SP>], #0]

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;
data = case memop of
  when MemOp_STORE
      data = X[t];
  when
      data = Mem[address, 1, address, dbytes, acctype] = data;
when
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtendAccType_ORDEREDMemOp_LOAD] = data;[data, regsize];

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(8) data;
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 = case memop of
  when MemOp_STORE
      data = X[t];
  when
      data = Mem[address, 1, address, dbytes, acctype] = data;
when
      data = Mem[address, dbytes, acctype];
      X[t] = ZeroExtendAccType_ORDEREDMemOp_LOAD] = data;[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.
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>]1, #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.

Operation

bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(n == 31);

if n == 31 then
  CheckSPAlignment();
  address = SP[4];
else
  address = X[n];

data = case memop of
  MemOp_STORE
    data = X[t];
  MemOp_LOAD
    data = Mem[address, dbytes, acctype] = data;
  when
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtend(AccType_ORDERED, MemOp_LOAD) = data, (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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size | opc |

32-bit (size == 10)

STLUR <Wt>, [<Xn|SP>{, #<simm>}

64-bit (size == 11)

STLUR <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);
integer datasize = 8 << scale;
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;

  if memop == MemOp_LOAD; then
    SetNotTagCheckedInstruction(is_load_store && n == 31);

  bits(64) address;
  bits(datasize) data;

  if n == 31 then
    if wback if n == t if n != 31 then
      ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
      case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
          assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        when Constraint_UNKNOWN wback != TRUE; // writeback is UNKNOWN
          rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNDEF rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_NOP EndOfInstruction();

    if memop == MemOp_STORE if wback if n == t if 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
          assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
          rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNDEF EndOfInstruction();
        when Constraint_NOP EndOfInstruction();

    if n == 31 then
      if memop == MemOp_PREFETCH then CheckSPAAlignment();
    else
      address = X[n];

    address = address + offset;
    if ! postindex then
      address = address + offset;

    data = case memop of
      when MemOp_STORE
        if rt_unknown then
          data = bits(datasize) UNKNOWN;
        else
          rt = X[t];
          Mem[address, datasize DIV 8, rt, acctype] = data;
        when
          data = Mem[address, datasize DIV 8, rt, acctype];
        if signed then
          X[t] = SignExtend(data, regsize);
        else
          X[t] = ZeroExtend(data, regsize);
        when
          MemOp_PREFETCHPrefetch(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;
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*.

### Unscaled offset

**STLURB \(<Wt>, [<Xn|SP>\{, \#<simm>\}]\)**

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>imm9</td>
<td>0 0</td>
</tr>
<tr>
<td>Rn</td>
<td>Rt</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.
- `<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' if 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_LOAD, MemOp_STORE} IN {MemOp_STORE, MemOp_PREFETCH};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

  bits(64) address;
  bits(datasize) data;

  if memop = MemOp_LOAD;
  SetNotTagCheckedInstruction(is_load_store && n == 31);

  bits(64) address;
  bits(8) data;

  if n == 31 then
    if wback && n == t && n != 31 then
      ConstrainUnpredictable(Unpredictable_WBOVERLAPPDL);
      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 rt_unknown = FALSE;        // value stored is original value
        when Constraint_NOP rt_unknown = TRUE;           // value stored is UNKNOWN
        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 rt_unknown = FALSE;        // value stored is original value
        when Constraint_NOP rt_unknown = TRUE;           // value stored is UNKNOWN
        EndOfInstruction();
    if n == 31 then
      if memop != MemOp_PREFETCH then CheckSPAlignment();
      address = SP[];
      else
        address = X[n];
      address = address + offset;
    if postindex then
      address = address + offset;

    data = case memop of
      when MemOp_STORE
        if rt_unknown then
          data = bits(datasize) UNKNOWN;
        else
          data = X[t];
        Mem[address, 1, address, datasize DIV 8, acctype] = data;
      when
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
          X[t] = SignExtend(data, regsize);
        else
          X[t] = ZeroExtend(data, regsize);
      when MemOp_PREFETCHPrefetch(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;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

```c
if (PSTATE.DIT == 1) {
    AccType_ORDERED[MemOp_LOAD] = data;
    Addr = address;
}
else {
    // Code for AccType_ORDERED[MemOp_LOAD]
    // Code for Addr
}
```
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Unscaled offset

STLURH <Wt>, [<Xn|SP]{, #<simm>}

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_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 = if opc<0> == '1' then MemOp_LOAD;
    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_LOAD, MemOp_STORE};
  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;
    SetNotTagCheckedInstruction(is_load_store && n == 31);

  bits(64) address;
  bits(16) data;

  if n == 31 then
    if wback && n == t && n != 31 then
      ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
      wback = FALSE;       // writeback is suppressed
      rt_unknown = TRUE;   // value stored is UNKNOWN
    when Constraint_UNKNOWN
      rt_unknown = TRUE;   // value stored is UNKNOWN
    when Constraint_UNDEF
      EndOfInstruction();

    if memop == MemOp_STORE && wback && n != t && n != 31 then
      ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
      rt_unknown = FALSE;  // value stored is original value
      rt_unknown = TRUE;   // value stored is UNKNOWN
      rt_unknown = TRUE;   // value stored is UNKNOWN
      rtUnknown = TRUE;    // value stored is UNKNOWN
    end case;

    if n == 31 then
      if memop != MemOp_PREFETCH then
        CheckSPAlignment();
      end if;
    end if;

    address = address + offset;
  else
    address = X[n];

    if !postindex then
      address = address + offset;
    end if;

    data = case memop of
      when MemOp_STORE
        if rt unknown then
          data = bits(datasize) UNKNOWN;
        else
          X[t];
    Mem[address, 2, address, datasize, acctype] = data;
      when
        data = Mem[address, datasize, acctype];
        if signed then
          X[t] = SignExtend(data, regsize);
        else
          X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCHPrefetch(address, t<4:0>);
      if wback then
        address = bits(64) UNKNOWN;
      elsif postindex then
        address = address + offset;
      if n == 31 then
        SP[] = 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.
STLXP

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.

Assembly 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;
bmp(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 then
  SetNotTagCheckedInstruction(is_load_store && n == 31);
if s == t || (pair && t == 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; // result is UNKNOWN
    when Constraint_NONE rt_unknown = FALSE; // store original value
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
if memop == MemOp_STORE then
  if s == 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];
if rt_unknown then
  data = bits(datasize) UNKNOWN;
else
  bits(datasize DIV 2) el1 = case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      elsif pair then
        data = if BigEndian() then el1:el2 else el2:el1;
      end;
    when MemOp_LOAD
      data = if BigEndian() then X[t] else X[t2];
  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() then el1:el2 else el2:el1;
else
  data = X[n];
  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, address, dbytes, acctype] = data;
status = Acctype.ORDEREDATOMIC(ExclusiveMonitorsStatus) = data;
status = Acctype.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));
  else
    data = Mem[address + 0, 8, acctype];
    X[t] = Mem[address + 8, 8, acctype];
    X[t2] = Mem[address + 9, 8, acctype];
    X[s] = [t] ZeroExtend(status, 32); (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.
STLXR

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 \(<Ws>, <Xt>, [<Xn|SP>,#,0]\)

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.

\(<Xt>\) 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 "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(elsize) data;
constant integer dbytes = elsize DIV 8;
bit(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  boolean is_load_store = MemOpLOAD IN {MemOp STORE IN {MemOp STORE,MemOp LOAD}};
  SetNotTagCheckedInstruction(is_load_store && n == 31);
if memop == MemOpLOAD;
  SetNotTagCheckedInstruction(is_load_store && n == 31);
if s == t then (pair || t == t2) then
  Constraint c = ConstraintUnpredictable(UnpredictableOVERLAP);
  assert c IN {Constraint_UNKNOWN,Constraint_UNDEF,Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF rt_unknown = FALSE; // store original value
    when Constraint_NOP EndOfInstruction();
if memop == MemOpSTORE then
  if s == t || (pair && s == t2) then
    Constraint c = ConstraintUnpredictable(UnpredictableDATAOVERLAP);
    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 EndOfInstruction();
if s == n && n != 31 then
  Constraint c = ConstraintUnpredictable(UnpredictableBASEOVERLAP);
  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 EndOfInstruction();
if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];
if rt unknown then
  data = bits(elsize) UNKNOWN;
else
  data = case memop of
    when MemOp STORE
      if rt unknown then
        data = bits(datasize) UNKNOWN;
      else pair then
        bits(datasize DIV 2) el1 = 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 bits(datasize DIV 2) el2 = el1
          data = if BigEndian() then el1 : el2 else el2 : el1;
        else
          data = X[t];
        // Check whether the Exclusives monitors are set to include the
        bit status = '1';
    when

// physical memory locations corresponding to virtual address
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, address, dbytes, acctype] = data;
  status = AccType_ORDEREDATOMIC(ExclusiveMonitorsStatus) = data;
  status = Error => 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>
    else
      X[t] = data<elsize-1:0>
    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));
    else
      data = Mem[address + 0, 8, acctype];
      X[t] = data<8:0, acctype>
      X[t2] = data<12:0, acctype>
    end
  end
  X[s] = ZeroExtend(status, 32);(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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 0  | Rs | 1  | (1) | (1) | (1) | (1) | Rn | 1  | (1) | (1) | (1) | (1) | Rt |

size   L  o0  Rs  1  (1)  (1)  (1)  (1)  Rn  1  (1)  (1)  (1)  (1)  Rt

No offset

STLXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = Integer t2 = UInt(Rs);    // ignored by all loads and store-release
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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEErr() then
    boolean is_load_store = true;
    memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);
    if memop == MemOp_STORE then
        SetNotTagCheckedInstruction(is_load_store && n == 31);
        if s == t then
            if 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();
            else
                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();
            end if
        else
            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();
                end if
        end if
    end if
else
    if rt_unknown then
        data = bits(8) UNKNOWN;
        data = case memop of
            when MemOp_STORE
                if rt_unknown then
                    data = bits(datasize) UNKNOWN;
                elsif pair then
                    bits(datasize DIV 2) el1 = 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
                        bits(datasize DIV 2) el2 = X[t];
                        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
if AArch64.ExclusiveMonitorsPass(address, l) then
// This atomic write will be rejected if it does not refer
// to the same physical locations after address translation, [address, dbytes) then
// This atomic write will be rejected if it does not refer
// to the same physical locations after address translation.
Mem[address, l](address, dbytes, acctype) = data;
status = AccType ORDEREDATOMIC[ExclusiveMonitorsStatus] = data;

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 dbyte-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
Y[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<datasize-1:128>; X[t2] = data<128-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));
else
X[t] = Mem[address + 0, 8, acctype];
X[t2] = Mem[address + 8, 8, acctype];
else
data = Mem[ExclusiveMonitorsStatus]();[address, dbytes, acctype];
X[s] = [s] ZeroExtend(status, 32);[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.
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>size</th>
<th>Rs0000</th>
<th>L</th>
<th>o001</th>
<th>Rt2</th>
</tr>
</thead>
</table>

No offset

STLXRH <Ws>, <Wt>, [<Xn|SP>/{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release
integer s = Integer(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); // ignored by all loads and store-release
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;
bv(16) data;
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 then
    SetNotTagCheckedInstruction(is_load_store && n == 31);
if s == t then
    if pair then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
            when Constraint_NONE   rt_unknown = FALSE;    // store original value
            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();
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                if pair then
                    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 bits(datasize DIV 2) el1 == X[t];
    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, 2) then
  // This atomic write will be rejected if it does not refer to the same physical locations after address translation.
  // This atomic write will be rejected if it does not refer to the same physical locations after address translation.
  Mem[address, 2, address, dbytes, acctype] = data;
  status = AccType_ORDEREDATOMIC[ExclusiveMonitorsStatus] = data;
  status[0] = 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));
      else
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
        X[s] = [t] ZeroExtend(status, 32);[data, regsize];
    else
      data = MemExclusiveMonitorsStatus[()];[address, dbytes, acctype];
      X[s] = ZeroExtend(status, 32);[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.
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>1 0 1 1 0 0 0 0</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></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}>]

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.

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

if opc == '11' then UNDEFINED;
integer scale = 2 + (Rt2); AccType accctype = 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
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
custom integer dbytes = datasize DIV 8;

boolean rt_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = MemOp STORE IN {MemOp STORE, MemOp LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);
    SetNotTagCheckedInstruction(is_load_store && n == 31);
if memop == MemOp LOAD; then
    SetNotTagCheckedInstruction(is_load_store && n == 31);
if n == 31 then if t == t2 then
    Constraint c = Constraint(UNpredictable|UNpredictable | NOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDER, Constraint NOP};
    case c of
        when Constraint UNKNOWN rt unknown = TRUE; // result is UNKNOWN
        when Constraint UNDER UNDEFINED;
        when Constraint NOP EndOfInstruction();
    if n == 31 then
        CheckSPAlignment();
        address = SP[];
        else
        address = X[n];
        address = address + offset;
        if !postindex then
            address = address + offset;
        endcase memop of
            when MemOp STORE
                data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, {address + 0, dbytes, acctype} = data1; AccType VECSTREAM =] = data1; address
            when MemOp LOAD
                data1 = Mem[address+dbytes, dbytes, {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
                    AccType VECSTREAM =] = data2;[n] = address;
```
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.

```
31 30 29 28 27 26 25 24 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 0 1 0 1 0 0 0 0 0 0 0 0 | imm7 | Rt2 | Rn | Rt |
```

32-bit (opc == 00)

```
STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

// Empty. 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);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + Int(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;
bots(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_STORE IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);
    if memop == MemOp_LOAD;
        SetNotTagCheckedInstruction(is_load_store && n == 31);

if n == 31 then
    if t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_INVALID);
        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();

        if n == 31 then
            CheckSPAlignment();
        address = SP[];
        else
            address = X[n];

        address = address + offset;
        if ! postindex then
            address = address + offset;

data1 = case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        data2 = if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address, dbytes, [address + 0 , dbytes, acctype] = data1;
    when MemOp_LOAD
        data1 = Mem[address+dbytes, dbytes, [address + 0 , dbytes, acctype];
        data2 = Mem[address+dbytes, dbytes, [address + 0 , 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
                if n == 31 then
                    SP[] = address;
                else
                    AccType STREAM[mem] = data2;[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></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>imm7</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</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>
</tr>
</tbody>
</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>
```

boolean wback = TRUE;
boolean postindex = TRUE;

### Pre-index

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>imm7</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</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>!
```

boolean wback = TRUE;
boolean postindex = FALSE;

### Signed offset

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>imm7</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

boolean wback = TRUE;
boolean postindex = TRUE;
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 = 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 "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

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(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); 
    if n == 31 && !wback then
        data1 = data1;  
        data2 = data2;
    if n == 31 then
        if 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_UNDER then UNDEFINED;
                when Constraint_NOP then EndOfInstruction();
            if n == 31 then
                CheckSPAlignment();
            else
                address = X[n];
            if !postindex then
                address = address + offset;
            if !postindex then
            
            datal = case memop of
                when MemOp_STORE
                    datal = V[t];
                data2 = V[t2];
            Mem[(address, dbytes, {address + 0, dbytes, acctype}) = data1; AccType_VEC(Mem) = data1];
            if rt unknown then
                ddata = bits(datasize) UNKNOWN;
                ddata2 = bits(datasize) UNKNOWN;
            Mem[(address + dbytes, dbytes, acctype}] = data2; AccType_VEC(Mem) = data2;
            data2 = ddata;
        if wback then
            if postindex then
                address = address + offset;
            if n == 31 then
                SP[] = address;
            else
                X[n] = address;
```
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | imm7 | Rt2 | Rn | Rt |

32-bit (opc == 00)

```c
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```c
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | imm7 | Rt2 | Rn | Rt |

32-bit (opc == 00)

```c
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>]
```

64-bit (opc == 10)

```c
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>]
```

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | imm7 | Rt2 | Rn | Rt |

32-bit (opc == 00)

```c
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>]
```

64-bit (opc == 10)

```c
STP <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 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);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + (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;
bite(64) offset = LSL(SignExtend(imm7, 64), scale);
```

STP
if HaveMTEExt() then
  if memop == MemOp_LOAD then
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);
    boolean wb_unknown = FALSE;
    if wback && (t == n || t2 == n) && n != 31 then
      Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
      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; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if n == 31 then
      if memop == MemOp_LOAD 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 !postindex then
        address = SP[n];
      else
        address = X[n];
    if !postindex then
      address = address + offset;
      if rt_unknown && t == n then
        data1 = bits(datasize) UNKNOWN;
      else
        data1 = 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, dbytes, [address + 0, dbytes, acctype] = data1; AccType_NORMAL] = data1;[address +
      when Memop_LOAD
        data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype]);
data2 = [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
  X[t] = data1;
  AccType_NORMALMem = data2;
[t?] = data2;
if wback then
  if postindex 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 | Rs |

**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;
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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 1 | 1 | 1 | 1 | 0 | 0 | x | 0 | 0 | imm9 | 1 | 1 | Rn | Rs |

opc

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
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 = FALSE;
Integer scale = UInt(opc<1>:size);
If scale > 4 then UNDEFINED;
Bits(64) offset = SignExtend(imm9, 64);

---

Unsigned offset

<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>Rn</td>
<td>imm12</td>
</tr>
<tr>
<td>Rt</td>
<td>opc</td>
</tr>
</tbody>
</table>

---

8-bit (size == 00 && opc == 00)

    STR <Bt>, [<Xn|SP>{, #<pimm}>]

16-bit (size == 01 && opc == 00)

    STR <Ht>, [<Xn|SP>{, #<pimm}>]

32-bit (size == 10 && opc == 00)

    STR <St>, [<Xn|SP>{, #<pimm}>]

64-bit (size == 11 && opc == 00)

    STR <Dt>, [<Xn|SP>{, #<pimm}>]

128-bit (size == 00 && opc == 10)

    STR <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.

<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

```plaintext
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 AccType_VEC
            data = Mem[address, datasize DIV 8, AccType_VEC];
            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

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm9</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>
</tr>
<tr>
<td>size</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit (size == 10)

```c
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bv32 offset = SignExtend(imm9, 64);
```

#### 64-bit (size == 11)

```c
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bv64 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>1</td>
<td>x</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>
<td></td>
<td></td>
</tr>
<tr>
<td>size</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit (size == 10)

```c
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bv32 offset = SignExtend(imm9, 64);
```

#### 64-bit (size == 11)

```c
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bv64 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>1</td>
<td>x</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>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>
</tr>
<tr>
<td>size</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bv32 offset = SignExtend(imm9, 64);
32-bit (size == 10)

\[ \text{STR }<Wt>, \ [<Xn|SP>\{, \ #<pimm>\}] \]

64-bit (size == 11)

\[ \text{STR }<Xt>, \ [<Xn|SP>\{, \ #<pimm>\}] \]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = \text{UInt}(size);

bits(64) offset = \text{LSL} (\text{ZeroExtend}(\text{imm12}, 64), \text{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>** 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

```plaintext
integer n = \text{UInt}(Rn);
integer t = \text{UInt}(Rt);

integer datasize = 8 \ll scale; \text{AccType acctype = AccType_NORMAL; 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 \ll scale;
```

STR (immediate)
if 

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; 
SetNotTagCheckedInstruction(is_load_store & & n == 31 & & !wback); 

bits(64) address; 
bits(datasize) data; 

boolean rt_unknown = FALSE; 

if wback & & n == t & & n != 31 then 
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 wback = 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 n == 31 then 

if memop == MemOp_PREFETCH then CheckSPAlignment(); 
else 
address = SP[]; 

if !postindex then 
if ! postindex then 
address = address + offset; 

if rt_unknown then 
data = bits(datasize) UNKNOWN; 
else 
data = case memop of 
when MemOp_STORE 
if rt_unknown then 
data = bits(datasize) UNKNOWN; 
else 
data = X[t]; 
Mem[address, datasize DIV 8, address, datasize DIV 8, acctype] = data; 
when 
data = Mem[address, datasize DIV 8, acctype]; 
if signed then 
X[t] = SignExtend(data, regsize); 
else 
X[t] = ZeroExtend(data, regsize); 
when MemOp_PREFETCHPrefetchAccType_NORMALMemOp_LOAD = data; 
(address, t<4:0>); 

if wback then
if postindex 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>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
<th>option</th>
<th>S</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</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 "Rm" 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.
For the 8-bit variant: 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>

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 accctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
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);
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, address, datasize DIV 8, acctype] = data;
    when AccType_VEC
        [t] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, address, datasize DIV 8, acctype], AccType_VEC][]{t} = data;
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        [t] = address;
    else
        V[t] = data;[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)

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.

32-bit (size == 10)

\[
\text{STR} \ <Wt>, \ [<Xn>|<SP>], \ (<Wm>|<Xm>)\{, <extend> \{<amount>\}}
\]

64-bit (size == 11)

\[
\text{STR} \ <Xt>, \ [<Xn>|<SP>], \ (<Wm>|<Xm>)\{, <extend> \{<amount>\}}
\]

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);

integer datasize = 8 << scale;

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_PREFETCH};
    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 then
        SetNotTagCheckedInstruction(is_load_store && n == 31);
    bits(64) address;
    bits(datasize) data;
    if n == 31 then
        if wback then n == t && n == 31 then
            ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
            assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
            case c of
                when Constraint_WBSUPPRESS then wback = FALSE; // writeback is suppressed
                when Constraint_UNKNOWN then wb_unknown = TRUE; // writeback is UNKNOWN
                when Constraint_UNDEF then UNDEFINED;
                when Constraint_NOP then 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 then rt_unknown = FALSE; // value stored is original value
            when Constraint_UNKNOWN then rt_unknown = TRUE; // value stored is UNKNOWN
            when Constraint_UNDEF then UNDEFINED;
            when Constraint_NOP then EndOfInstruction();
    if n == 31 then
        if memop == MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
        else
            address = X[n];
            address = address + offset;
            if ! postindex then address = address + offset;
        data = case memop of
            when MemOp_STORE
                if rt_unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = X[t];
                Mem[address, datasize DIV 8, address, datasize DIV 8, acctype] = data;
            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

XAccType_NORMAL[newMemOp_LOAD] = data;[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></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm9</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>size</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>
</tr>
</tbody>
</table>

**STRB <Wt>, [<Xn|SP>], #<simm>

```java
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = Integer.scale = imm9(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

<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>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm9</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>size</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>
</tr>
</tbody>
</table>

**STRB <Wt>, [<Xn|SP>, #<simm>]

```java
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = Integer.scale = imm9(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

<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>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</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>
</tr>
</tbody>
</table>

**STRB <Wt>, [<Xn|SP>], #<pimm>]

```java
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = Integer.scale = imm12(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0); imm12, 64, scale=32
```

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.

<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
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;
```
Operation

STRB (immediate)
if `HaveMTEExt()` then
  boolean is_load_store = MemOp_STORE IN {MemOp_STORE, MemOp_LOAD};
  SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);
end;

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD;
  SetNotTagCheckedInstruction(is_load_store & n == 31 & !wback);

bits(64) address;
bits(8) data;

boolean rt_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 wbback = FALSE;       // writeback is suppressed
    when Constraint_UNKNOWN  wb unknown = TRUE;      // writeback is UNKNOWN
    when Constraint_UNDEF    UNDEFINED;
    when Constraint_NOP      EndOfInstruction();
  end;
end;

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;
end;

if n == 31 then
  if memop == MemOp_PREFETCH then CheckSPAlignment();
  else
    address = SP[];
  end;
  address = X[n];
if !postindex then
  address = address + offset;
if rt_unknown then
  data = bits(8) UNKNOWN;
else
  data = case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
    Mem[address, 1, address, datasize DIV 8, accotype] = data;
    when
      data = Mem[address, datasize DIV 8, accotype];
      if signed then
        X[t] = SignExtend(data, regsize);
      else
        X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
      PrefetchAccType_NORMAL MemOp_LOAD = data;
  end;
end;
if wback then
if postindex 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)

\[
\text{STRB } <\text{Wt}>, [<\text{Xn}|\text{SP}>], (<\text{Wm}|<\text{Xm}>), \langle\text{extend}\rangle \langle\text{amount}\rangle]
\]

Shifted register (option == 011)

\[
\text{STRB } <\text{Wt}>, [<\text{Xn}|\text{SP}>], <\text{Xm}>, \langle\text{LSL }<\text{amount}\rangle\rangle]
\]

Assembler Symbols

\(<\text{Wt}>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

\(<\text{Wm}>\) When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

\(<\text{Xm}>\) When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

\(<\text{extend}>\) Is the index extend specifier, encoded in “option”:

\[
\begin{array}{c|c}
\text{option} & \text{<extend>} \\
010 & \text{UXTW} \\
110 & \text{SXTW} \\
111 & \text{SXTX} \\
\end{array}
\]

\(<\text{amount}>\) 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;
bits(64) offset = ExtendReg(m, extend_type, 0);
(extend_type, shift);
if HaveMTEExt() then
  boolean is_load_store = (memop == MemOp_STORE)
  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;
  SetNotTagCheckedInstruction(is_load_store & n == 31);

bits(64) address;
bits(8) data;
if n == 31 then if wback & n == t & n == 31 then
  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 rt unknown = FALSE;        // value stored is original value
    when Constraint_NOP rt_unknown = TRUE;           // value stored is UNKNOWN
    EndOfInstruction();
  end;
if memop == MemOp_STORE & wback & n == t & n == 31 then
  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 rt_unknown = TRUE;         // value stored is UNKNOWN
    when Constraint_NOP EndOfInstruction();
  end;
if n == 31 then if memop != MemOp_PREFETCH then CheckSPAlignment();
else
  address = SP[];
  address = address + offset;
  if ! postindex then
    address = address + offset;
  end;

data = case memop of
  when MemOp_STORE
    if rt unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
    Mem[address, 1, address, datasize DIV 8, acctype] = data;
  when
    data = Mem[address, datasize DIV 8, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCHPrefetch(address, t<4:0>);
  in else
    if wback unknown then
      address = bits(64) UNKNOWN;
    elsif postindex then
      address = address + offset;
    end;
  if n == 31 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.
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

<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>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Post-index

```java
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = integer scale =  signextend(size);
bits(64) offset =  signextend(imm9, 64);
```

### 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>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Pre-index

```java
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = integer scale =  signextend(size);
bits(64) offset =  signextend(imm9, 64);
```

### Unsigned 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>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Unsigned offset

```java
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale =  signextend(size);
bits(64) offset =  lsl(zerorecorded_scale(imm12, 64), 1);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.

<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);

    address = SP[];
    data = bits(16) UNKNOWN;
    if rt_unknown then
        data = case memop of
            MemOp_STORE then X[t];
            MemOp_PREFETCH then CheckSPAlignment();
            else
                data = case memop of
                    MemOp_PREFETCH then PrefetchAccType_NORMAL;
                    else
                        data = Mem[address, 2, address, datasize DIV 8, acctype] = data;
                        data = Mem[address, address, address, datasize DIV 8, acctype] = data;
                        data = X[t] = SignExtend(data, regsize);
                        data = X[t] = ZeroExtend(data, regsize);
                        data = Mem[MemOp_PREFETCH_PREFETCHAccType_NORMAL_memop, Load] = data;
    end;

    if wback then
        STRH (immediate)
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 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  |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**32-bit**

\[
\text{STRH} <Wt>, [<Xn>|SP>, (<Wm>|<Xm>)], <extend> {<amount>}
\]

- \( <Wt> \): 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>

---

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.
- \( <Wm> \) is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- \( <Xm> \) 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.
- \( <amount> \) 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":

```c
if option<1> == '0' then UNDEFINED; // sub-word index
boolean wback = 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 1 else 0; integer shift = if S == '1' then scale else 0;
```
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;
end

integer datasize = 8 << scale;
if HaveMTEExt() then
  boolean is_load_store = MemOp_STORE IN {MemOp_STORE, MemOp_LDM}
  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 then
  SetNotTagCheckedInstruction(is_load_store && n == 31);

bits(64) address;
bits(16) data;
if n == 31 then
  if wback if n == t && n == 31 then
    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];
  end if
  address = address + offset;
  if !postindex then
    address = address + offset;
  end if
  data = case memop of
    MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      else
        data = X[t];
      end if
      Mem[address, 2] = Mem[address, datasize DIV 8, acctype] = data;
      if signed then
        X[t] = SignExtend(data, regsize);
      else
        X[t] = ZeroExtend(data, regsize);
      end if
      if MemOp_PREFETCH then
        Prefetch(address, t<4:0>);
      end if
      if wback then
        if wb_unknown then
          address = bits(64) UNKNOWN;
        elsif postindex then
          address = address + offset;
        if n == 31 then
          SP[] = address;
        else
          ...
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

```plaintext
else
  AccType_NORMAL MemOp_LOAD] = data; [n] = address;
```
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)

\[
\text{STTR } \langle Wt \rangle, [\langle Xn\rvert SP \rangle\{, \#<\text{simm}>\}]
\]

64-bit (size == 11)

\[
\text{STTR } \langle Xt \rangle, [\langle Xn\rvert SP \rangle\{, \#<\text{simm}>\}]
\]

```python
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.
- \(<Xn\rvert SP>\): Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- \(<\text{simm}>\): 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;

integer datasize = 8 << scale;
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:
    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;
        SetNotTagCheckedInstruction(is_load_store && n == 31);
    
    bits(64) address;
    bits(datasize) data;

    if n == 31 then if wback || n != t || n != 31 then
        ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
        case c of
            when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
            when Constraint_UNKNOWN wb unknown = TRUE; // writeback is UNKNOWN
            when Constraint_UNDEF
                rt unknown = FALSE;  // value stored is original value
            when Constraint_NOP
                rt unknown = TRUE;   // value stored is UNKNOWN
                UNDEFINED;
        
        old wback = FALSE;
        if wback then
            rt unknown = FALSE;  // value stored is original value
        
        if n == 31 then
            if memop != MemOp_PREFETCH then
                CheckSPAlignment();
            else
                address = X[n];
            
            address = address + offset;
            if postindex then
                address = address + offset;
        
        data = 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_PREFETCHPrefetch(address, t<4:0>);

        if wback then
            address = bits(64) UNKNOWN;
            if postindex then
                address = address + offset;
            if n == 31 then
                SP[] = address;
else
       \[\text{address, data size DIV 8, acctype} = \text{data}]\][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.
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.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>imm9</th>
<th>Gn</th>
<th>Gr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</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>
</tr>
</tbody>
</table>

**Unscaled offset**

STTRB `<Wt>`, `<Xn|SP>{, #<simm>`}

```plaintext
bits(64) offset = 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_LOAD, MemOp_STORE} || MemOp_PREFETCH;
    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;
        SetNotTagCheckedInstruction(is_load_store && n == 31);

    bits(64) address;
    bits(8) data;

    if n == 31 then
        if wb_unknown && n == t && n != 31 then
            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 rt unknown = FALSE;  // value stored is original value
                when Constraint_NOP rt unknown = TRUE;   // value stored is UNKNOWN
                EndOfInstruction();

            if memop == MemOp_STORE if wb_unknown && n == t && n != 31 then
                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 original value
                    when Constraint_UNDEF rt unknown = FALSE;  // value stored is original value
                    when Constraint_NOP rt unknown = TRUE;   // value stored is UNKNOWN
                    EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then
            CheckSPAlignment();
        else
            address = SP[];
        address = address + offset;
    if ! postindex then
        address = address + offset;

    data = 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
                t2 = signExtend(data, regsize);
            else
                t2 = zeroExtend(data, regsize);
            X[t2] = t2;
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wb_unknown then
        if n == 31 then
            SP[] = 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 Effect value of PSTATE.UAO is 0 and either:
• The instruction is executed at EL1.
• The instruction is executed at EL2 when the Effect 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}>]
```

```
bits(64) offset = 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_PREFETCH};
    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;
        SetNotTagCheckedInstruction(is_load_store && n == 31);
    bits(64) address;
    bits(16) data;
    if n == 31 then
        if wback && n != 31 then
            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 rt unknown = FALSE;        // value stored is original value
            when Constraint_NOP EndOfInstruction();
        if memop == MemOp_STORE && wback && n != 31 then
            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 EndOfInstruction();
        if n == 31 then
            if memop != MemOp_PREFETCH then CheckSPAlignment();
            address = SP[];
            else
                address = X[n];
            address = address + offset;
        if !postindex then
            address = address + offset;
        data = 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;
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>0x0</th>
<th>0x0</th>
<th>0x0</th>
<th>0x0</th>
<th>x</th>
<th>0</th>
<th>0</th>
<th>imm9</th>
<th>0x0</th>
<th>0x0</th>
<th>Rn</th>
<th>Rt</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>
</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}>]

**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**

```plaintext
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);  // SetNotTagCheckedInstruction(is_load_store)

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if !postindex then
    address = address + offset;

    case memop of
    when MemOp_STORE
        data = Y[t];
        Mem[address, datasize DIV 8, address, datasize DIV 8, acctype] = data;
    when AccType_VEC = data;
    when MemOp LOAD
        data = Mem[address, datasize DIV 8, address, datasize DIV 8, acctype];
        AccType_VEC](t) = data;
    if wback then
        if postindex then
            address = address + offset;
        if n == 31 then
            [f] = address;
        else
            VEC[t] = data; 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.

32-bit (size == 10)

STUR <Wt>, [<Xn|SP>], #<simm>>

64-bit (size == 11)

STUR <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

integer n = UInt(Rn);
integer t = UInt(Rt);
integer datasize = 8 << scale;
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;
        SetNotTagCheckedInstruction(is_load_store && n == 31);

    bits(64) address;
    bits(datasize) data;

    if n == 31 then
        if n == t && n != 31 then
            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 rt unknown = FALSE;       // value stored is original value
                when Constraint_NOP EndOfInstruction();
            end_case;

    if memop == MemOp_STORE if wback if 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 EndOfInstruction();
            when Constraint_NOP EndOfInstruction();
        end_case;

    if n == 31 then
        if memop == MemOp_PREFETCH then CheckSPAisignment();
        else
            address = SP[];
            address = address + offset;
            if ! postindex then
                address = address + offset;
        end_case;

        data = case memop of
            when MemOp_STORE
                if rt unknown then
                    data = bits(datasize) UNKNOWN;
                else
                    data = X[t];

                Mem(address, datasize DIV 8, address, datasize DIV 8, acctype) = data;
            when
                data = Mem(address, datasize DIV 8, acctype);
                if signed then
                    data = SignExtend(data, regsize);
                else
                    data = ZeroExtend(data, regsize);
            end_case;

            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;
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.

Unscaled offset

STURB <Wt>, [<Xn|SP>{, #<simm>}

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_NORMAL;
MemOp memop = MemOp_STORE;
boolean signed = FALSE;
integer regsize = if size == '11' then 64 else 32;
if opc0 == '1' then
  // store or zero-extending load
  memop = if opc0 == '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 opc0 == '1' then UNDEFINED;
  else
    // sign-extending load
    memop = MemOp_LOAD;
    if size == '10' & opc0 == '1' then UNDEFINED;
    regsize = if opc0 == '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;
    SetNotTagCheckedInstruction(is_load_store && n == 31);
  
  bits(64) address;
  bits(8) data;
  if n == 31 then
    if wb_unknown then
      address = bits(64) UNKNOWN;
    elsif postindex then
      address = address + offset;
  else
    address = address + offset;
  
  data = case memop of
    MemOp_STORE: if rt_unknown then
      data = bits(datasize) UNKNOWN;
    else
      data = X[t];
    Mem(address, 1, address, datasize DIV 8, acctype) = data;
    when
      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

XAccType_NORMAL[MemOp_LOAD] = data; \( n \) = 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.
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.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Unscaled offset

STURH <Wt>, [<Xn|SP>], #<simm>]

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 accotype = AccType_NORMAL;
MemOp memop = if opc<0> == '1' then MemOp_PREFETCH else MemOp_STORE;
integer regsize = if size == '11' then 64 else 32;
signed = FALSE;
if size == '11' then
    memop = if opc<0> == '1' then MemOp_PREFETCH else MemOp_PREFETCH;
else
    // sign-extending load
    memop = if opc<0> == '1' then UNDEFINED else 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);

    bits(64) address;
    bits(datasize) data;
    if memop == MemOp_LOAD;
        SetNotTagCheckedInstruction(is_load_store && n == 31);

    bits(64) address;
    bits(16) data;
    if n == 31 then
        if wback && n == 31 then
            ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
            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 rt unknown = FALSE;  // value stored is original value
                when Constraint_NOP rt unknown = TRUE;   // value stored is UNKNOWN
        else
            rt unknown = FALSE;  // value stored is original value
        end;
        CaseOf wback = FALSE;       // writeback is suppressed
        Constraint_WBSUPPRESS, Constraint_UNKNOWN,Constraint_UNDEF, Constraint_NOP;
        c = Unpredictable_WBOVERLAPLD;
        case c of
            when Constraint_WBSUPPRESS rt unknown = FALSE;  // value stored is original value
            when Constraint_UNKNOWN rt unknown = TRUE;   // value stored is UNKNOWN
            when Constraint_UNDEF rt unknown = FALSE;  // value stored is original value
            when Constraint_NOP rt unknown = TRUE;   // value stored is UNKNOWN
        end;
        if n == 31 then
            if memop != MemOp_PREFETCH then
                CheckSPAlignment();
            end;
            address = SP[];
            elsif postindex then
                address = address + offset;
                data = case memop of
                    when MemOp_STORE
                        if rt unknown then
                            data = bits(datasize) UNKNOWN;
                        else
                            data = X[t];
                    end;
                    Mem[address, 2];
                    if signed then
                        X[t] = SExt(data, regsize);
                    else
                        X[t] = ZExt(data, regsize);
                    end;
                    when MemOp_PREFETCH
                        Prefetch(address, t<4:0>);
                    end;
                    if wback then
                        address = bits(64) UNKNOWN;
                    elsif postindex then
                        address = address + offset;
                end;
                if n == 31 then
                    SP[] = 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.
STXP

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.

32-bit (sz == 0)

STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

STXP <Wa>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

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;
bite(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  if memop == MemOp_LOAD then
    SetNotTagCheckedInstruction(is_load_store && n == 31);
    if memop = MemOp_STORE then
      if s == t || (pair && t == t2) then
        Case
          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();
          end Case;
      end Case;
    end if;
    if s == n && n != 31 then
      Case
        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();
        end Case;
    end if;
    if n == 31 then
      CheckSPAlignment();
      address = SP[];
    elif rn_unknown then
      address = bits(64) UNKNOWN;
    else
      address = X[n];
    end if;
  if rt_unknown then
    data = bits(datasize) UNKNOWN;
  else
    bits(datasize DIV 2) el1 = case memop of
      when MemOp_STORE
        if rt_unknown then
          data = bit(datasize) UNKNOWN;
        elsif pair then
          bits(datasize DIV 2) el1 = X[t];
        end if;
      else
        if BigEndian() then el1:el2 else el2:el1;
      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 el1 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 = Acctype ATOMICExclusiveMonitorsStatus = data;
status &= [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));
    else
      data = Mem[address + 0, 8, acctype];
      X[t] = Mem[address + 8, 8, acctype];
      X[s] = ZeroExtend(status, 32);[data, regsize];
  else // elsize == 64
  ...
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.

<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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**size**

32-bit (size == 10)

STXR <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STXR <Ws>, <Xt>, [<Xn|SP>{,#0}]

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release
integer elsize = 8 << UInt(size); // ignored by load/store single register
integer s = UInt(size); // 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;
```

For information about the CONSTRANDED 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 "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(elsize) data;
constant integer dbytes = elsize DIV 8;
bit[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_LOAD, MemOp_STORE} && n == 31);
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == MemOp_LOAD();
  SetNotTagCheckedInstruction(is_load_store && n == 31);

if s == t then	s | pair | t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
  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];

if rt_unknown then
  data = bits(elsize) UNKNOWN;
else
  data = case memop of
    when MemOp_STORE
      if rt_unknown then
        data = bits(datasize) UNKNOWN;
      elsif pair then
        bits(datasize DIV 2) el1 = X[t];
      bit status = '1';
    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
      bits(datasize DIV 2) el2 = X[t];
      else
        data = if BigEndian() then all : el2 else el2 : all;
      else
        data = X[t];
      bit status = '1';
    // Check whether the Exclusives monitors are set to include the

STXR
// 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, address, dbytes, acctype] = data;
  status = Acctype_ATOMICExclusiveMonitorsStatus = data;
  status = 4;
end;

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[t1] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
      else
        X[t1] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
      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));
      else
        data = Mem[address, 8, acctype];
        X[t1] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
        X[s] = ZeroExtend(status, 32);(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.
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.

No offset

STXRB <Ws>, <Wt>, [<Xn|SP>{,0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release
integer t2 = UInt(Rs); // 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;

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.
• <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.
bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    boolean is_load_store = if memop IN {MemOp_STORE, MemOp_LOAD} then
        SetNotTagCheckedInstruction(is_load_store && n == 31);
else
    if memop == MemOp_STORE then
        SetNotTagCheckedInstruction(is_load_store && n == 31);
if s == t then if pair || t == 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; // result is UNKNOWN
    when Constraint_NONE rt_unknown = FALSE; // store original value
    when Constraint_UNDEF 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 rtUnknown = TRUE; // store UNKNOWN value
    when Constraint_NONE rtUnknown = 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 rnUnknown = TRUE; // address is UNKNOWN
    when Constraint_NONE rnUnknown = FALSE; // address is original base
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
if n == 31 then
    CheckSPAlignment();
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) s1 = X[t];
        else
            data = case memop of
                when MemOp_STORE
                    if rt_unknown then
                        data = bits(datasize) UNKNOWN;
                    elsif pair then
                        bits(datasize DIV 2) s1 = X[t];
                else
                    data = if BigEndian() then s1 : s2 else s2 : s1;
                end;
        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
    bits(datasize DIV 2) s1 = X[t];
else
    data = if BigEndian() then s1 : s2 else s2 : s1;
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
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
  // This atomic write will be rejected if it does not refer
  // to the same physical locations after address translation. [address, dbytes) then
  // This atomic write will be rejected if it does not refer
  // to the same physical locations after address translation.
  Mem[address, 1] [address, dbytes, acctype] = data;
  status = AccType_ATOMICExclusiveMonitorsStatus = data;
end
status = ()
  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 dbyte-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 = MemExclusiveMonitorsStatus() [address, dbytes, acctype];
        X[s] = s = ZeroExtend(status, 32) [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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 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 L o0 Rt2

No offset

STXRH \(<Ws>, <Wt>, [<Xn|SP>,\{,#0\}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs);  // ignored by all loads and store-release
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 regsize = if elsize == 64 then 64 else 32;
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(16) data;
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 then
    SetNotTagCheckedInstruction(is_load_store & n == 31);
if s == t then 
    if pair then s == t then
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF};
        case c of
            when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
            when Constraint_NONE rt_unknown = FALSE; // store original value
            when Constraint_UNDEF // store UNKNOWN value
            EndOfInstruction();
if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        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 // store original base
                // store original value
            EndOfInstruction();
if s == n && n != 31 then
    if n == 31 then
        CheckSPAlignment();
extif rn_unknown then
    address = bits(64) UNKNOWN;
extelse
    address = X[n];
if rt_unknown then
    data = bits(16) UNKNOWN;
extelse
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
etif pair then
            bits(datasize DIV 2) el1 = 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 BigEndian() then el1 : el2 else el2 : el1;
extelse
    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, 2) then
  // This atomic write will be rejected if it does not refer
  // to the same physical locations after address translation. [address, dbytes] then
  // This atomic write will be rejected if it does not refer
  // to the same physical locations after address translation.
Mem[address, 2], [address, dbytes, acctype] = data;
  status = Acctype_ATOMICExclusiveMonitorsStatus = data;
status = 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
    // 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 = MemExclusiveMonitorsStatus[address, dbytes, acctype];
      X[s] = s = ZeroExtend(status, 32);(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)

<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>1 1 0 1 1 0 0 1 1 1 1 imm9 0 1 Xn</td>
</tr>
</tbody>
</table>

Pre-index
(ARMv8.5)

<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>1 1 0 1 1 0 0 1 1 1 1 imm9 1 1 Xn</td>
</tr>
</tbody>
</table>

Signed offset
(ARMv8.5)

<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>1 1 0 1 1 0 0 1 1 1 1 imm9 1 0 Xn</td>
</tr>
</tbody>
</table>

Signed offset

STZ2G [<Xn|SP>, #<simm>]
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;
```
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)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)

**Post-index**

STZG [<Xn|SP>], #<simm>

```plaintext
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)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)

**Pre-index**

STZG [<Xn|SP>, #<simm>]

```plaintext
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  | 0  | imm9| 1  | 0  | Xn | (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)| (1)

**Signed offset**

STZG [<Xn|SP>{, #<simm>}

```plaintext
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

```
bits(64) address;
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(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;
```
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>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
<th>option</th>
<th>imm3</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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>

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

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'.

SUB (extended register)
<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 "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;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);

bit carry_in =
operand2 = NOT(operand2); (result, -) = if sub_op then
                                          operand2 = NOT(operand2);
                                          carry_in = '1';
                                      else
                                          carry_in = '0';
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);
if d == 31 then 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.

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

```plaintext
32-bit (sf == 0)

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

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

```plaintext
64-bit (sf == 1)

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

```plaintext
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(imm12:12, 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;
bits(datasize) operand2 = imm;
bit carry_in;
operand2 = NOT(imm);
(result, -) = if sub_op then
    operand2 = NOT(operand2);
else
    operand2 = '1';
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
operand1, operand2, carry_in;
if d == 31 then 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.

(old) htmldiff from- (new)
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).

32-bit (sf == 0)

```
31 30 29 28 27 26 25 24 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
```

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

64-bit (sf == 1)

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

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
integer shift_amount = UInt(imm6);
```

```
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>NEG (shifted register)</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>
Operation

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

bit carry_in;

operand2 = NOT(operand2);
(result, -) = if sub_op then
                operand2 = NOT(operand2);
                carry_in = '1';
            else
                carry_in = '0';
            (result, nzcv) = AddWithCarry(operand1, operand2, '1');
            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.

(old) htmldiff from- (new)
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)

|   |   |   |   |   | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |   | 0 | 0 | 0 | 1 | 1 | 0 |   | 0 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Integer
SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>

```plaintext
d = UInt(Xd);
n = UInt(Xn);

bits(4) tag_offset = uimm4;
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);

boolean ADD = FALSE;

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

```plaintext
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, uimm4, exclude);
else
    rtag = '0000';

(result, -) = if ADD then
    (result, -) = AddWithCarry(operand1, NOT(offset), '1');
else
    (result, -) = AddWithCarry(operand1, NOT(offset), '1');

result = AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```
SUBP

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  | 0  | 1  | 1  | 0  | Xm | 0  | 0  | 0  | 0  | 0  | Xn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Integer

SUBP <Xd>, <Xn|SP>, <Xm|SP>

integer d = UInt(Xd);
n = UInt(Xn);
m = UInt(Xm); Boolean setflags = FALSE;

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.

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, -) = AddWithCarry(operand1, operand2, '1');
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | Xm | 0  | 0  | 0  | 0  | 0  | Xn | 0  | Xd |

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>$S == '1' &amp;&amp; Xd == '111111'$</td>
</tr>
</tbody>
</table>

Operation

```plaintext
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
boolean setflags = TRUE;

integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
boolean setflags = TRUE;

integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
boolean setflags = TRUE;

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');
PSTATE.<N,Z,C,V> = nzcv;if setflags then;
PSTATE.<N,Z,C,V> = nzcv;if setflags then;
X[d] = result;
```

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

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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>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>
<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>}}
```

```python
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>UXTL</td>
</tr>
<tr>
<td>010</td>
<td>LSLL</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 <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”:

<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>UTXH</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.

### 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) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;

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

(result, nzcv) = AddWithCarry(operand1, operand2, '1');
operand1, operand2, carry_in;
PSTATE.<N,Z,C,V> = nzcv;
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>0</th>
<th>1</th>
<th>!= 1x</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>
| op | S | shift

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

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

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

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

```plaintext
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>&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**

- **CMP (immediate)** is preferred when **Rd == '11111'**
Operation

```c
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;
operand2 = NOT(imm);
if sub_op then
    operand2 = NOT(operand2);
else
    carry_in = '1';

(result, nzcv) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcv;
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)` and `NEGS`.

### 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><code>CMP (shifted register)</code></td>
<td>Rd == '11111'</td>
</tr>
<tr>
<td><code>NEGS</code></td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>
Operation

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

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

(result, nzcv) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

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

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.

```
1 1 0 1 0 0 0 0
   mm16
```

System

SVC #<imm>

```
// Empty. 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(imm16); (imm);
```

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

\[
\begin{array}{cccccccccccccccccccccc}
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & A & R & 1 & | & 1 & 0 & 0 & 0 & 0 & 0 & 0 & | & R & n & | & R & t
\end{array}
\]

**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>]

```plaintext
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 <<

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.
bits(64) address;
bits(8) data;
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, 1, ldacctype];
Mem[address, datasize DIV 8, ldacctype] = X[s];

X[t] = ZeroExtend(data, 32);
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  | A  | R  | 1  | Rs | 1  | 0  | 0  | 0  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |

size

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

bits(64) address;
bits(16) data;
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, 2, ldacctype];
Mem[address, datasize DIV 8, ldacctype] = X[s];
Mem[address, datasize DIV 8, stacctype] = X[s];
X[t] = ZeroExtend(data, 32); data, regsize);
SYS

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 1  | 0  | op1 | CRn | CRm | op2 | Rt |
| L  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

System

SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}

AACrh64.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

<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>CRn == '0111' &amp;&amp; CRm == '100x' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_AT</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 == '100'</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 == '100'</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

if has_result then

X[t] = AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else

AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, sys_op0, sys_op1, sys_crn, sys_crm, sys_op2,

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

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.

### 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

```assembly
if has_result then
    X[t] = AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys.crm, sys_op2, Rt, L);
else
    AArch64.SysInstr(sys_op0, sys_op1, sys_crn, sys.crm, sys_op2, X(l, sys_op1, sys_crn, sys.crm, sys_op2);
```

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.

Two register table (len == 01)

\begin{verbatim}
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
\end{verbatim}

Three register table (len == 10)

\begin{verbatim}
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
\end{verbatim}

Four register table (len == 11)

\begin{verbatim}
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
\end{verbatim}

Single register table (len == 00)

\begin{verbatim}
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
\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{verbatim}
  Q   <Ta>
  0   8B
  1   16B
  \end{verbatim}
- \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
  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
  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 \(<R><t>, \#<imm>, <label>\)

```
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”:

```
\begin{array}{c|c}
  b5 & \<R> \\
  0 & W \\
  1 & X \\
\end{array}
```

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> == op then 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: 2018-09-13T13:25Z

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

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.

<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>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 1 0 0 0</td>
<td>Rm 0 len 1 0 0</td>
</tr>
</tbody>
</table>

Two register table (len == 01)

\[
\text{TBX } <\text{Vd}>. <\text{Ta}>, \{ <\text{Vn}>.16B, <\text{Vn}+1>.16B \}, <\text{Vm}>.<\text{Ta}>
\]

Three register table (len == 10)

\[
\text{TBX } <\text{Vd}>. <\text{Ta}>, \{ <\text{Vn}>.16B, <\text{Vn}+1>.16B, <\text{Vn}+2>.16B \}, <\text{Vm}>.<\text{Ta}>
\]

Four register table (len == 11)

\[
\text{TBX } <\text{Vd}>. <\text{Ta}>, \{ <\text{Vn}>.16B, <\text{Vn}+1>.16B, <\text{Vn}+2>.16B, <\text{Vn}+3>.16B \}, <\text{Vm}>.<\text{Ta}>
\]

Single register table (len == 00)

\[
\text{TBX } <\text{Vd}>. <\text{Ta}>, \{ <\text{Vn}>.16B \}, <\text{Vm}>.<\text{Ta}>
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } Q == '1' \text{ then } 128 \text{ else } 64; \\
\text{integer } \text{elements} &= \text{datasize} \text{ DIV } 8; \\
\text{integer } \text{regs} &= \text{UInt}(\text{len}) + 1; \\
\text{boolean } \text{is_tbl} &= (\text{op} == '0');
\end{align*}
\]

Assembler Symbols

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

\(<\text{Ta}>\) Is an arrangement specifier, encoded in “Q”:

\[
\begin{array}{c|c}
Q & \text{<Ta>} \\
\hline
0 & 8B \\
1 & 16B
\end{array}
\]

\(<\text{Vn}>\) 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.

\(<\text{Vn}+1>\) Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

\(<\text{Vn}+2>\) Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

\(<\text{Vn}+3>\) Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

\(<\text{Vm}>\) 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;
in integer index;

// Create table from registers
for i = 0 to regs-1
 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
 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.

<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>b5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>b40</td>
<td>imm14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

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> == op then if operand<bit_pos> == bit_val then BranchTo(PC[] + offset, BranchType_DIR);
```

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

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)

TSB CSYNC

```c

systemHintOp if !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_BT.

BTypeCompatible = BTypeCompatible_BT() then (op2<2:1>)

otherwise EndOfInstruction();(); // Instruction executes as NOP
```

TSB CSYNC
TraceSynchronizationBarrier(); case op of
  when SystemHintOp_YIELD
    when SystemHintOp_WFE
      if !InterruptPending() then
        ClearEventRegister();
      else
        if PSTATE.RI = $1 then
          // Check for traps described by the OS which may be EL1 or EL2.
          AArch64.CheckForWFeTrap(S1, TRUE);
          if EL2Enabled() && PSTATE.RI IN ($10, $11) && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFeTrap(S1, TRUE);
          if HaveEL(SL) && PSTATE.RI = $13 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFeTrap(S1, TRUE);
        WaitForEvent();
      when SystemHintOp_WFI
        if !InterruptPending() then
          if PSTATE.RI = $11 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch64.CheckForWFeTrap(S1, FALSE);
          if EL2Enabled() && PSTATE.RI IN ($10, $11) && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFeTrap(S1, FALSE);
          if HaveEL(SL) && PSTATE.RI = $13 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFeTrap(S1, FALSE);
        WaitForInterrupt();
      when SystemHintOp_SEV
        SendEvent();
      when SystemHintOp_SEVL
        SendEventLocal();
      when SystemHintOp_ESB
        SynchronizeErrors();
    AAchr64.ESBOperation();
    if EL2Enabled() && PSTATE.RI IN {EL0, EL1} then
      AAchr64.vESBOperation();
    TakeUnmaskedErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BT
    BTypeNext = '0',
  otherwise // do nothing

UABA

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 CPACR_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

UABA <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.
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.

```
<table>
<thead>
<tr>
<th>Rm</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

Three registers, not all the same type

UABAL2 <Vd>, <Ta>, <Vn>, <Vm>

```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 accumulate = (op == '0');
boolean unsigned = (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></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.
UABD

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 \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{UABD <Vd>,<T>, <Vn>,<T>, <Vm>,<T>}

integer \(d\) = \texttt{UInt}(Rd);
integer \(n\) = \texttt{UInt}(Rn);
integer \(m\) = \texttt{UInt}(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer datasize = if \(Q\) == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (\(U\) == '1');
boolean accumulate = (\(ac\) == '1');

Assemble 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":

<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>

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

Operation

\textbf{CheckFPAdvSIMDEnabled64}();
bits(datasize) operand1 = \texttt{V}[n];
bits(datasize) operand2 = \texttt{V}[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then \texttt{V}[d] else \texttt{Zeros}();
for e = 0 to elements-1
    element1 = \texttt{Int}(\texttt{Elem}[operand1, e, esize], unsigned);
    element2 = \texttt{Int}(\texttt{Elem}[operand2, e, esize], unsigned);
    absdiff = \texttt{Abs}(element1-element2)<esize-1:0>;
    \texttt{Elem}[result, e, esize] = \texttt{Elem}[result, e, esize] + absdiff;
\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.
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.

Three registers, not all the same type

UABDL{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 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 QTQ</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>8B</td>
</tr>
<tr>
<td>00 1</td>
<td>16B</td>
</tr>
<tr>
<td>01 0</td>
<td>4H</td>
</tr>
<tr>
<td>01 1</td>
<td>8H</td>
</tr>
<tr>
<td>10 0</td>
<td>2S</td>
</tr>
<tr>
<td>10 1</td>
<td>4S</td>
</tr>
<tr>
<td>11 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;
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>; (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.
UADALP

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.

```
31 30 29 28 27 26 25 24 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 0 0 0 0 0 1 1 0 1 0 Rn Rd
```

Vector

UADALP <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);
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.
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 \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}
0  Q 1 0 1 1 0  size 1 0 0 0 0 0 1 0 1 0  Rn  Rd
U op

Vector

UADDLP <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');
\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 “size:Q”:

\begin{verbatim}
size Q <Ta>
00 0  4H
00 1  8H
01 0  2S
01 1  4S
10 0  1D
10 1  2D
11 x  RESERVED
\end{verbatim}

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

\texttt{<Tb>} Is an arrangement specifier, encoded in “size:Q”:

\begin{verbatim}
size Q <Tb>
00 0  8B
00 1  16B
01 0  4H
01 1  8H
10 0  2S
10 1  4S
11 x  RESERVED
\end{verbatim}
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.
UBFM

Unsigned 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}.

<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>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>
</tr>
</tbody>
</table>

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

\texttt{UBFM <Wd>, <Wn>, #<immr>, #<imms>}

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

\texttt{UBFM <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);
(wmask, tmask) = DecodBitMasks(N, immrs, immr, FALSE);
```

Assembler Symbols

\(<\text{Wd}>\) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<\text{Wn}>\) is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

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

\(<\text{Xn}>\) is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

\(<\text{immr}>\) is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

\(<\text{imms}>\) 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>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**

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

// perform bitfield move on low bits
bits(datasize) bot = [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, R) AND wmask; dst<8> else dst;

// combine extension bits and result bits
X[d] = bot AND tmask; 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.
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.

<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>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmode</td>
<td>opcode</td>
<td>scale</td>
<td>Rn</td>
<td>Rd</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)

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;
  if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

  rounding = case opcode<2:1>\rrmode 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(FPCR);
    otherwise
      UNDEFINED;
  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.
<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;

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, TRUE, FPCR, rounding);(intval, fracbits, unsigned, FPCR, rounding);
V[d] = fltval;

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 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.
32-bit to half-precision (sf == 0 && type == 11)
(ARMv8.2)

UCVTF <Hd>, <Wn>

32-bit to single-precision (sf == 0 && type == 00)

UCVTF <Sd>, <Wn>

32-bit to double-precision (sf == 0 && type == 01)

UCVTF <Dd>, <Wn>

64-bit to half-precision (sf == 1 && type == 11)
(ARMv8.2)

UCVTF <Hd>, <Xn>

64-bit to single-precision (sf == 1 && type == 00)

UCVTF <Sd>, <Xn>

64-bit to double-precision (sf == 1 && type == 01)

UCVTF <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'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
when '11'
    if HaveFP16Ext() then
        fltsize = 16;
    else
        UNDEFINED;
rounding = 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'    // MOV 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 !HaveFPJCVTZSExt() then UNDEFINED; 
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS(FPCR);
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;

intval = case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[n];
    fltval = [d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, TRUE, FPCR, rounding);
        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 = [d] = intval;
        X[n];
        fltval = FPToFixedJS(fltval, FPCR, TRUE);
        X[d] = ZeroExtend[d] = fltval<intval<31:0>, 64>;
```

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

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

32-bit (sf == 0)

```
UDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
UDIV <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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
integer datasize = if sf == '1' then 64 else 32;
boolean unsigned = (o1 == '0');
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;
if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(operand1, unsigned)) / Real(Int(operand2, unsigned));
X[d] = result<datasize-1:0>;
```
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 `CPACR_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)**

```
|   |   |   |   |   |   |   |   | size |   | M |   | Rm |   |   |   |   | 1 | 1 | 0 | H | 0 |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   | 0   | Q | 1 | 0 | 1 | 1 | 1 | 1 |   | size | L | M |   | Rm |   |   |   |   | 1 | 1 | 1 | 0 | H | 0 |   |   |   |   |   |   |   |   |
```

**Vector**

```
UDOT <Vd>..<Ta>, <Vn>..<Tb>, <Vm>.4B..<index>]
```

```
if !HaveDOTPeXt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
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”:
  - `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 "M:Rm" fields.
- `<index>` Is the element index, encoded in the "H:L" fields.
Operation

```plaintext
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]);
        end if
        res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```
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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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 | 0  | Rm | 1  | 0  | 0  | 1  | 0  | 1  | Rn | 1  | 0  | 0  | 1  | 0  | 1  | Rd |

Three registers of the same type

UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

\[
\text{if } \text{!HaveDOTPExt()} \text{ then UNDEFINED}; \\
\text{if } \text{size }\neq '10' \text{ then UNDEFINED}; \\
\text{boolean signed } = (U == '0'); \\
\text{if } \text{size }\neq '10' \text{ then UNDEFINED}; \\
\text{boolean signed } = (U == '0'); \\
\text{integer } d = \text{UInt}(Rd); \\
\text{integer } n = \text{UInt}(Rn); \\
\text{integer } m = \text{UInt}(Rm); \\
\text{integer } esize = 8 \ll \text{UInt}(\text{size}); \\
\text{integer } \text{datasize } = \text{if } Q == '1' \text{ then } 128 \text{ else } 64; \\
\text{integer } \text{elements } = \text{datasize } \div \text{esize};
\]

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”:

\[
\begin{array}{c|c|c}
Q & <Ta> \\
0 & 0 \ 8S \\
1 & 1 \ 16S \\
\end{array}
\]

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

\(<Tb>\)  Is an arrangement specifier, encoded in “Q”:

\[
\begin{array}{c|c|c}
Q & <Tb> \\
0 & 0 \ 8B \\
1 & 1 \ 16B \\
\end{array}
\]

\(<Vm>\)  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]);
        end
        res = res + element1 * element2;
    end
    Elem[result, e, esize] = Elem[result, e, esize] + res;
end
V[d] = result;
```

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  |

U   00

64-bit

Assembling 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>UMULL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation:

```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');

integer result;
result = if sub_op then
    result = Int(operand3, TRUE) + (Int(operand3, unsigned) - (Int(operand1, TRUE) * 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.
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 idxds = 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”: 
### 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.
UMLAL, UMLAL2 (vector)

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>

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | Rn | Rd | 01 |

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.
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 \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}
31 30 29 28 27 26 25 24 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 1 size L M Rm 0 1 1 0 H 0 Rn Rd o2
\end{verbatim}

Vector

\texttt{UMLSL[2]} <\texttt{Vd}>, <\texttt{Ta}>, <\texttt{Vn}>, <\texttt{Ts}>[\texttt{<index>}] 

\begin{verbatim}
integer idxdsize = if H == '1' then 128 else 64;
native integer index;
bit Rmhi;
\textbf{case size of}
\begin{itemize}
\item when '01' index = \texttt{UInt}(H:L:M); Rmhi = '0';
\item when '10' index = \texttt{UInt}(H:L); Rmhi = M;
\item otherwise UNDEFINED;
\end{itemize}

integer d = \texttt{UInt}(Rd);
native integer n = \texttt{UInt}(Rn);
native integer m = \texttt{UInt}(Rmhi:Rm);

integer esize = 8 << \texttt{UInt}(size);
native integer datasize = 64;
native integer part = \texttt{UInt}(Q);
native integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
\end{verbatim}

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":

\begin{verbatim}
Q 2
0 [absent]
1 [present]
\end{verbatim}

\texttt{<Vd>} \hspace{1cm} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<Ta>} \hspace{1cm} Is an arrangement specifier, encoded in “size”:

\begin{verbatim}
size <Ta>
00 RESERVED
01 4S
10 2D
11 RESERVED
\end{verbatim}

\texttt{<Vn>} \hspace{1cm} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\texttt{<Tb>} \hspace{1cm} 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>Q</th>
<th>&lt;Vm&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>

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();
b_bits(datasize) operand1 = Vpart[n, part];
b_bits(idxdsize) operand2 = V[m];
b_bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
b_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.

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.
UMSUBL

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 <Xd>, <Wn>, <Wm>, <Xa>

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 minuend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMNEGL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

```
bits(32) operand1 = bits(datasize) operand1 = X[n];
bits(32) operand2 = bits(datasize) operand2 = X[m];
bits(64) operand3 = bits(destsize) operand3 = X[a];

integer result;

result = if sub_op then
    result = Int(operand3, TRUE) - (operand3, unsigned) - (Int(operand1, TRUE) * (operand1, unsigned) * Int(operand2, unsigned));
else
    result = Int(operand1, 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.
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.

<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>Rm</td>
<td>0</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>Rd</td>
<td>Ra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<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

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 datasize = 64;
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(64) operand1 = bits(datasize) operand1 = X[n];
bits(64) operand2 = bits(datasize) operand2 = X[m];
integer result;
result = Int(operand1, TRUE) * Int(operand1, unsigned) * Int(operand2, TRUE);(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.

Vector

UMULL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```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 = 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>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”:
<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) 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.
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.

Three registers, not all the same type

UMULL[2] <Vd>,<Ta>,<Vn>,<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 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) 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.
**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 *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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 1  | Rn |  | Rd |
```

**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**

`<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;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**

```
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>;result, e, esize] = (element1+element2+1)<esize:1>;
V[d] = result;
```
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.

\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
1 1 0 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 0 1 1 1 1
\end{verbatim}

\begin{verbatim}
CRm  op2
\end{verbatim}

System

\begin{verbatim}
WFE

\textbf{SystemHintOp} op;

\textbf{case} CRm:op2 of

\textbf{when} '0000 000' op = \textbf{SystemHintOp_NOP};
\textbf{when} '0000 001' op = \textbf{SystemHintOp_YIELD};
\textbf{when} '0000 010' op = \textbf{SystemHintOp_WFE};
\textbf{when} '0000 011' op = \textbf{SystemHintOp_WFI};
\textbf{when} '0000 100' op = \textbf{SystemHintOp_SEV};
\textbf{when} '0000 101' op = \textbf{SystemHintOp_SEVL};
\textbf{when} '0000 111'
SEE "XPACLRI";
\textbf{when} '0001 xxx'
SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
\textbf{when} '0010 000'
\textbf{if} ! HaveRASExt() then \textbf{EndOfInstruction}();                  // Instruction executes as NOP
op = \textbf{SystemHintOp_ESB};
\textbf{when} '0010 001'
\textbf{if} ! HaveStatisticalProfiling() then \textbf{EndOfInstruction}();    // Instruction executes as NOP
op = \textbf{SystemHintOp_PSB};
\textbf{when} '0010 010'
\textbf{if} ! HaveSelfHostedTrace() then \textbf{EndOfInstruction}();         // Instruction executes as NOP
op = \textbf{SystemHintOp_TSB};
\textbf{when} '0010 100'
op = \textbf{SystemHintOp_CSDB};
\textbf{when} '0011 xxx'
SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
\textbf{when} '0100 xx0'
op = \textbf{SystemHintOp_BTI};
BTypeCompatible = \textbf{BTypeCompatible_BTI} (op2<2:1>);
\textbf{otherwise} \textbf{EndOfInstruction}();  // Instruction executes as
\end{verbatim}
if case op of
when SystemHintOp_YIELD() then
    Hint_Yield();
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 then
        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_SEVSendEvent() then
        when SystemHintOp_SEVSendEventLoc() then
        when SystemHintOp_ESBSynchronizeError() then
            AArch64.ESBOperation();
            if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
                AArch64.vESBOperation();
                TakeUnmaskedErrorInterrupts();
        when SystemHintOp_PSB then
            ProfilingSynchronizationBarrier();
        when SystemHintOp_TSB then
            TraceSynchronizationBarrier();
        when SystemHintOp_CSDB then
            ConsumptionOfSpeculativeDataBarrier();
        when SystemHintOp_BTI then
            BTypeNext = '00';
        otherwise // do nothing
    end case

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

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.

### System

```
<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0</td>
<td></td>
</tr>
</tbody>
</table>
```

WFI
if Case of
  when SystemHintOp_YIELD
    Yield();
  when SystemHintOp_WFE
    if EventRegSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTraps(EL1, TRUE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTraps(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTraps(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.CheckForWfxTraps(EL1, FALSE);
      if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTraps(EL2, FALSE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTraps(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();
    TakeUnmaskedErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    BitTypeNext = '00';
  otherwise // do nothing

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | (0)| (0)| (0)| (0)| 0  | 0  | 1  | 1  | 1  | 1  | 1  |

CRm

System

XAFlag

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;

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

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

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</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>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XPACD (D == 1)

```
XPACD <Xd>
```

XPACI (D == 0)

```
XPACI <Xd>

    boolean data = (D == '1');
    integer d = Uint(Rd);
    if integer n = Uint(Rn);
    if !HavePACExt() then
        UNDEFINED; UNDEFINED;
    if n != 31 then UNDEFINED;
```

### System (ARMv8.3)

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

System

```
XPACLRI

    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);
```
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.
case op of
    when SystemHintOp_YIELDHint_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(EL2) && PSTATE_EL == EL2 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(EL2) && PSTATE_EL == EL2 then
                                // Check for traps described by the Secure Monitor.
                                AArch64.CheckForWFeTrap(EL3, FALSE);
                            WaitForInterrupt();
                        when SystemHintOp_SEVSendEvent() =>
                        when SystemHintOp_SEVLSendEventLoc() =>
                        when SystemHintOp_ESBSynchronizeError() =>
                            AArch64.ESBOperation();
                            if EL2Enabled() && PSTATE_EL IN {EL0, EL1} then
                                AArch64.vESBOperation();
                                TakeUnmaskedErrorInterrupts();
                        when SystemHintOp_PSBProfilingSynchronizationBarrier() =>
                        when SystemHintOp_TSBTraceSynchronizationBarrier() =>
                        when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier() =>
                        when SystemHintOp_BTI BTypeNext = '00';
                    otherwise // do nothing