Proprietary Notice

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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).

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

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

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

ADR: Form PC-relative address.

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

AND (immediate): Bitwise AND (immediate).

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

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

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

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

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

ASRV: Arithmetic Shift Right Variable.

AT: Address Translate: an alias of SYS.

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

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

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

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

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.

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: PSTATE.C Flag Inversion.

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 zero bits.

CMN (extended register): Compare Negative (extended register): an alias of ADDS (extended register).

CMN (immediate): Compare Negative (immediate): an alias of ADDS (immediate).

CMN (shifted register): Compare Negative (shifted register): an alias of ADDS (shifted register).

CMP (extended register): Compare (extended register): an alias of SUBS (extended register).

CMP (immediate): Compare (immediate): an alias of SUBS (immediate).

CMP (shifted register): Compare (shifted register): an alias of SUBS (shifted register).

CNEG: Conditional Negate: an alias of CSNEG.

CRC32B, CRC32H, CRC32W, CRC32X: CRC32 checksum.

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.

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.

HINT: Hint instruction.

HLT: Halt instruction.

HVC: Hypervisor Call.

IC: Instruction Cache operation: an alias of SYS.

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.

LDARB: Load-Acquire Register Byte.

LDARH: Load-Acquire Register Halfword.

LDAXP: Load-Acquire Exclusive Pair of Registers.

LDAXR: Load-Acquire Exclusive Register.

LDAXRB: Load-Acquire Exclusive Register Byte.

LDAXRH: Load-Acquire Exclusive Register Halfword.

LDCLR, LDCLRA, LDCLRAL, LDCLRL: Atomic bit clear on word or doubleword in memory.

LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB: Atomic bit clear on byte in memory.

LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH: Atomic bit clear on halfword in memory.

LDEOR, LDEORA, LDEORAL, LDEORL: Atomic exclusive OR on word or doubleword in memory.

LDEORB, LDEORAB, LDEORALB, LDEORLB: Atomic exclusive OR on byte in memory.

LDEORH, LDEORAH, LDEORALH, LDEORLH: Atomic exclusive OR on halfword in memory.

LDLAR: Load LOAcquire Register.

LDLARB: Load LOAcquire Register Byte.

LDLARH: Load LOAcquire Register Halfword.

LDNP: Load Pair of Registers, with non-temporal hint.

LDP: Load Pair of Registers.

LDPSW: Load Pair of Registers Signed Word.

LDR (immediate): Load Register (immediate).

LDR (literal): Load Register (literal).

LDR (register): Load Register (register).

LDRAA, LDRAB: Load Register, with pointer authentication.

LDRB (immediate): Load Register Byte (immediate).

LDRB (register): Load Register Byte (register).

LDRH (immediate): Load Register Halfword (immediate).

LDRH (register): Load Register Halfword (register).

LDRSB (immediate): Load Register Signed Byte (immediate).

LDRSB (register): Load Register Signed Byte (register).

LDRSH (immediate): Load Register Signed Halfword (immediate).

LDRSH (register): Load Register Signed Halfword (register).

LDRSW (immediate): Load Register Signed Word (immediate).

LDRSW (literal): Load Register Signed Word (literal).

LDRSW (register): Load Register Signed Word (register).

LDSET, LDSETA, LDSETAL, LDSETL: Atomic bit set on word or doubleword in memory.

LDSETB, LDSETAB, LDSETALB, LDSETLB: Atomic bit set on byte in memory.

LDSETH, LDSETAH, LDSETALH, LDSETLH: Atomic bit set on halfword in memory.

LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL: Atomic signed maximum on word or doubleword in memory.

LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB: Atomic signed maximum on byte in memory.

LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH: Atomic signed maximum on halfword in memory.

LDSMIN, LDSMINA, LDSMINAL, LDSMINL: Atomic signed minimum on word or doubleword in memory.

LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB: Atomic signed minimum on byte in memory.

LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH: Atomic signed minimum on halfword in memory.

LDTR: Load Register (unprivileged).

LDTRB: Load Register Byte (unprivileged).

LDTRH: Load Register Halfword (unprivileged).

LDTRSB: Load Register Signed Byte (unprivileged).

LDTRSH: Load Register Signed Halfword (unprivileged).

LDTRSW: Load Register Signed Word (unprivileged).

LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL: Atomic unsigned maximum on word or doubleword in memory.

LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB: Atomic unsigned maximum on byte in memory.

LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH: Atomic unsigned maximum on halfword in memory.

LDUMIN, LDUMINA, LDUMINAL, LDUMINL: Atomic unsigned minimum on word or doubleword in memory.

LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB: Atomic unsigned minimum on byte in memory.

LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH: Atomic unsigned minimum on halfword in memory.

LDUR: Load Register (unscaled).

LDURB: Load Register Byte (unscaled).

LDURH: Load Register Halfword (unscaled).

LDURSB: Load Register Signed Byte (unscaled).

LDURSH: Load Register Signed Halfword (unscaled).

LDURSW: Load Register Signed Word (unscaled).

LDXP: Load Exclusive Pair of Registers.

LDXR: Load Exclusive Register.

LDXRB: Load Exclusive Register Byte.

LDXRH: Load Exclusive Register Halfword.

LSL (immediate): Logical Shift Left (immediate): an alias of UBFM.

LSL (register): Logical Shift Left (register): an alias of LSLV.

LSLV: Logical Shift Left Variable.

LSR (immediate): Logical Shift Right (immediate): an alias of UBFM.

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

LSRV: Logical Shift Right Variable.

MADD: Multiply-Add.

MNEG: Multiply-Negate: an alias of MSUB.

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

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

MOV (register): Move (register): an alias of ORR (shifted register).

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).

NEG (shifted register): Negate (shifted register): an alias of SUB (shifted register).

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

NGC: Negate with Carry: an alias of SBC.

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

NOP: No Operation.

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

ORR (immediate): Bitwise OR (immediate).

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

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

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

PACGA: Pointer Authentication Code, using Generic key.

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

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

PRFM (immediate): Prefetch Memory (immediate).

PRFM (literal): Prefetch Memory (literal).

PRFM (register): Prefetch Memory (register).

PRFM (unscaled offset): Prefetch Memory (unscaled offset).

PSB CSYNC: Profiling Synchronization Barrier.

PSSBB: Physical Speculative Store Bypass Barrier.

RBIT: Reverse Bits.

RET: Return from subroutine.

RETAA, 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.

SBC: Subtract with Carry.

SBCS: Subtract with Carry, setting flags.

SBFIZ: Signed Bitfield Insert in Zero: an alias of SBFM.

SBFM: Signed Bitfield Move.

SBFX: Signed Bitfield Extract: an alias of SBFM.

SDIV: Signed Divide.

SETF8, SETF16: Evaluation of 8 or 16 bit flag values.

SEV: Send Event.

SEVL: Send Event Local.

SMADDL: Signed Multiply-Add Long.

SMC: Secure Monitor Call.

SMNEGL: Signed Multiply-Negate Long: an alias of SMSUBL.

SMSUBL: Signed Multiply-Subtract Long.

SMULH: Signed Multiply High.

SMULL: Signed Multiply Long: an alias of SMADDL.

SSBB: Speculative Store Bypass Barrier.

STADD, STADDL: Atomic add on word or doubleword in memory, without return: an alias of LDADD, LDADDA, LDADDAL, LDADDL.

STADDB, STADDLB: Atomic add on byte in memory, without return: an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB.

STADDH, STADDLH: Atomic add on halfword in memory, without return: an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH.

STCLR, STCLRL: Atomic bit clear on word or doubleword in memory, without return: an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL.

STCLRB, STCLRLB: Atomic bit clear on byte in memory, without return: an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.

STCLRH, STCLRLH: Atomic bit clear on halfword in memory, without return: an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.

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.

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.

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

SUB (immediate): Subtract (immediate).

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

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

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

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

SVC: Supervisor Call.

SWP, SWPA, SWPAL, SWPL: Swap word or doubleword in memory.

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

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

SXTB: Signed Extend Byte: an alias of SBFM.

SXTH: Sign Extend Halfword: an alias of SBFM.

SXTW: Sign Extend Word: an alias of SBFM.

SYS: System instruction.

SYSL: System instruction with result.

TBNZ: Test bit and Branch if Nonzero.

TBZ: Test bit and Branch if Zero.

TLBI: TLB Invalidate operation: an alias of SYS.

TSB CSYNC: Trace Synchronization Barrier.

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

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

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

UBFM: Unsigned Bitfield Move.

UBFX: Unsigned Bitfield Extract: an alias of UBFM.

UDIV: Unsigned Divide.

UMADDL: Unsigned Multiply-Add Long.

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

UMSUBL: Unsigned Multiply-Subtract Long.

UMULH: Unsigned Multiply High.

UMULL: Unsigned Multiply Long: an alias of UMADDL.

UXTB: Unsigned Extend Byte: an alias of UBFM.

UXTH: Unsigned Extend Halfword: an alias of UBFM.

WFE: Wait For Event.

WFI: Wait For Interrupt.

XPACD, XPACI, XPACLRI: Strip Pointer Authentication Code.

YIELD: YIELD.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

A64 -- SIMD and Floating-point Instructions (alphabetic order)

ABS: Absolute value (vector).

ADD (vector): Add (vector).

ADDHN, ADDHN2: Add returning High Narrow.

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

ADDP (vector): Add Pairwise (vector).

ADDV: Add across Vector.

AESD: AES single round decryption.

AESE: AES single round encryption.

AESIMC: AES inverse mix columns.

AESMC: AES mix columns.

AND (vector): Bitwise AND (vector).

BCAX: Bit Clear and XOR.

BIC (vector, immediate): Bitwise bit Clear (vector, immediate).

BIC (vector, register): Bitwise bit Clear (vector, register).

BIF: Bitwise Insert if False.

BIT: Bitwise Insert if True.

BSL: Bitwise Select.

CLS (vector): Count Leading Sign bits (vector).

CLZ (vector): Count Leading Zero bits (vector).

CMEQ (register): Compare bitwise Equal (vector).

CMEQ (zero): Compare bitwise Equal to zero (vector).

CMGE (register): Compare signed Greater than or Equal (vector).

CMGE (zero): Compare signed Greater than or Equal to zero (vector).

CMGT (register): Compare signed Greater than (vector).

CMGT (zero): Compare signed Greater than zero (vector).

CMHI (register): Compare unsigned Higher (vector).

CMHS (register): Compare unsigned Higher or Same (vector).

CMLE (zero): Compare signed Less than or Equal to zero (vector).

CMLT (zero): Compare signed Less than zero (vector).

CMTST: Compare bitwise Test bits nonzero (vector).

CNT: Population Count per byte.

DUP (element): Duplicate vector element to vector or scalar.

DUP (general): Duplicate general-purpose register to vector.

EOR (vector): Bitwise Exclusive OR (vector).

EOR3: Three-way Exclusive OR.

EXT: Extract vector from pair of vectors.

FABD: Floating-point Absolute Difference (vector).

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

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

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

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

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

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

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

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

FCADD: Floating-point Complex Add.

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

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

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

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

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

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

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).

FMINNMV: 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).

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.

MOV (vector): Move vector: an alias of ORR (vector, register).

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.

RAX1: Rotate and Exclusive OR.

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

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

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

REV64: Reverse elements in 64-bit doublewords (vector).

RSHRN, RSHRN2: Rounding Shift Right Narrow (immediate).

RSUBHN, RSUBHN2: Rounding Subtract returning High Narrow.

SABA: Signed Absolute difference and Accumulate.

SABAL, SABAL2: Signed Absolute difference and Accumulate Long.

SABD: Signed Absolute Difference.

SABDL, SABDL2: Signed Absolute Difference Long.

SADALP: Signed Add and Accumulate Long Pairwise.

SADDL, SADDL2: Signed Add Long (vector).

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.

SMLAL, SMLAL2 (by element): Signed Multiply-Add Long (vector, by element).

SMLAL, SMLAL2 (vector): Signed Multiply-Add Long (vector).

SMLSL, SMLSL2 (by element): Signed Multiply-Subtract Long (vector, by element).

SMLSL, SMLSL2 (vector): Signed Multiply-Subtract Long (vector).

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

SMULL, SMULL2 (by element): Signed Multiply Long (vector, by element).

SMULL, SMULL2 (vector): Signed Multiply Long (vector).

SQABS: Signed saturating Absolute value.

SQADD: Signed saturating Add.

SQDMLAL, SQDMLAL2 (by element): Signed saturating Doubling Multiply-Add Long (by element).

SQDMLAL, SQDMLAL2 (vector): Signed saturating Doubling Multiply-Add Long.

SQDMLSL, SQDMLSL2 (by element): Signed saturating Doubling Multiply-Subtract Long (by element).

SQDMLSL, SQDMLSL2 (vector): Signed saturating Doubling Multiply-Subtract Long.

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

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

SQDMULL, SQDMULL2 (by element): Signed saturating Doubling Multiply Long (by element).

SQDMULL, SQDMULL2 (vector): Signed saturating Doubling Multiply Long.

SQNEG: Signed saturating Negate.

SQRDMLAH (by element): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

SQRDMLAH (vector): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

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

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

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

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

SQRSHL: Signed saturating Rounding Shift Left (register).

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

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

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

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

SQSHLU: Signed saturating Shift Left Unsigned (immediate).

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

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

SQSUB: Signed saturating Subtract.

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).

SSHL: Signed Shift Left (register).

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

SSHR: Signed Shift Right (immediate).

SSRA: Signed Shift Right and Accumulate (immediate).

SSUBL, SSUBL2: Signed Subtract Long.

SSUBW, SSUBW2: Signed Subtract Wide.

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

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

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

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

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

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

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

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

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

STP (SIMD&FP): Store Pair of SIMD&FP registers.

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

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

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

SUB (vector): Subtract (vector).

SUBHN, SUBHN2: Subtract returning High Narrow.

SUQADD: Signed saturating Accumulate of Unsigned value.

SXTL, SXTL2: Signed extend Long: an alias of SSHLL, SSHLL2.

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.

UMLAL, UMLAL2 (by element): Unsigned Multiply-Add Long (vector, by element).

UMLAL, UMLAL2 (vector): Unsigned Multiply-Add Long (vector).

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

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

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

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

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

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).

URSQRTE: 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.

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

UZP1: Unzip vectors (primary).

UZP2: Unzip vectors (secondary).

XAR: Exclusive OR and Rotate.

XTN, XTN2: Extract Narrow.

ZIP1: Zip vectors (primary).

ZIP2: Zip vectors (secondary).

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ADC

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

```
ADC <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;

(result, -) =if sub_op then
    operand2 = NOT(operand2);

(result, nzcv) = 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.

ADCS

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

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
sf	0	1	1	1	0	1	0	0	0	0	Rm	0	0	0	0	0	0	Rn	Rd
	op	S																	

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

ADD (extended register)

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

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

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

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

```
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;

(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 d == 31 thenif 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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).

31	30	29	28	27	26	25	24	23–22	21–10	9–5	4–0
sf	0	0	1	0	0	0	1	shift	imm12	Rn	Rd
	op	S									

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12:(imm12 :Zeros(12), datasize);
    when '1x' 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”:

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

Alias Conditions

Alias	Is preferred when
MOV (to/from SP)	`shift == '00' && imm12 == '000000000000' && (Rd == '11111' || Rn == '11111')`

Operation

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

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

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

if d == 31 thenif 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ADD (shifted register)

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

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	0	1	0	1	1	shift	0	Rm	imm6	Rn	Rd
	op	S											

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

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”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<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) 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	size		1	1	0	0	0	1	1	0	1	1	1	0	Rn					Rd				

Advanced SIMD

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

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

if size != '11' then ReservedValue();

integer esize = 8 << UInt(size);
integer datasize = esize * 2;integer datasize = esize * 2;
integer elements = 2;ReduceOp op = ReduceOp_ADD;
```

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

<T> Is the source arrangement specifier, encoded in “size”:

size	<T>
0x	RESERVED
10	RESERVED
11	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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 CMN (extended register).

31	30	29	28	27	26	25	24	23	22	21	20-16	15-13	12-10	9-5	4-0
sf	0	1	0	1	0	1	1	0	0	1	Rm	option	imm3	Rn	Rd
	op	S													

32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;~~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 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 first source general-purpose register or stack pointer, encoded in the "Rn" field.

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

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

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

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

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

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

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

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

Alias	Is preferred when
CMN (extended register)	`Rd == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;

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

(result, nzcv) = AddWithCarry(operand1, operand2, '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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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).

31	30	29	28	27	26	25	24	23 22	21 … 10	9 … 5	4 … 0
sf	0	1	1	0	0	0	1	shift	imm12	Rn	Rd
	op	S									

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12:(imm12 :Zeros(12), datasize);
    when '1x' 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”:

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

Alias Conditions

Alias	Is preferred when
CMN (immediate)	Rd == '11111'

Operation

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

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

(result, nzcv) = AddWithCarry(operand1, 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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).

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	1	0	1	0	1	1	shift	0	Rm	imm6	Rn	Rd
	op	S											

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 ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

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”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<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

Alias	Is preferred when
CMN (shifted register)	`Rd == '11111'`

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');
(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ADDV

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

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

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

Advanced SIMD

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

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

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();

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

Assembler Symbols

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

size	<V>
00	B
01	H
10	S
11	RESERVED

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

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

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

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

ADR

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

31	30 29	28	27	26	25	24	23 … 5	4 … 0
0	immlo	1	0	0	0	0	immhi	Rd
op								

Literal

```
ADR <Xd>, <label>
```

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

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

Assembler Symbols

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

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

Operation

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

~~if page then~~
~~    base<11:0> = ~~

~~Zeros(12);~~

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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 … 5	4 … 0
1	immlo	1	0	0	0	0	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	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	0	1	1	0	Rn					Rd				
																			D												

Advanced SIMD

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UnallocatedEncoding();();
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));(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

AESE

AES single round encryption.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9-5	4-0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	0	0	1	0	Rn	Rd
																			D				

Advanced SIMD

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UnallocatedEncoding();();
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

AESIMC

AES inverse mix columns.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9-5	4-0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	1	1	1	0	Rn	Rd
																			D				

Advanced SIMD

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

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

Assembler Symbols

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

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

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

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

AESMC

AES mix columns.

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

Advanced SIMD

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

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

Assembler Symbols

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

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

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

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

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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
0	Q	0	0	1	1	1	0	0	0	1	Rm					0	0	0	1	1	1	Rn					Rd				
								size																							

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

boolean invert = (size<0> == '1');LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AN
```

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

```
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

AND (immediate)

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

31	30	29	28	27	26	25	24	23	22	21–16	15–10	9–5	4–0
sf	0	0	1	0	0	1	0	0	N	immr	imms	Rn	Rd
	opc												

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

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then~~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~~ ReservedValue();
(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 AND imm;
if d == 31 thencase 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	0	1	0	1	0	shift	0	Rm	imm6	Rn	Rd
	opc								N				

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~~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 ~~ReservedValue();

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<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);

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;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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).

31	30	29	28	27	26	25	24	23	22	21-16	15-10	9-5	4-0
sf	1	1	1	0	0	1	0	0	N	immr	imms	Rn	Rd
	opc												

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' thenboolean 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 ReservedValue();
(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

Alias	Is preferred when
TST (immediate)	`Rd == '11111'`

Operation

```
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~~
~~    whenLogicalOp_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):'00';~~[] = 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).

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

32-bit (sf == 0)

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

64-bit (sf == 1)

```
ANDS <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~~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 ~~ReservedValue();

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<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

Alias	Is preferred when
TST (shifted register)	`Rd == '11111'`

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
    whenLogicalOp_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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

B.cond

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

31	30	29	28	27	26	25	24	23–5	4	3–0
0	1	0	1	0	1	0	0	imm19	0	cond

19-bit signed PC-relative branch offset

```
B.<cond> <label>
```

```
bits(64) offset = SignExtend(imm19:'00', 64);~~(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~~(condition) then~~
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

B

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.

31	30	29	28	27	26	25–0
0	0	0	1	0	1	imm26
op						

26-bit signed PC-relative branch offset

```
B <label>
```

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

Assembler Symbols

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

Operation

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

BranchTo(PC[] + offset, BranchType_JMP);[] + offset, branch_type);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

[ISA_v84A_A64_xml_00bet7 (old)] htmldiff from-ISA_v84A_A64_xml_00bet7 [(new) ISA_v84A_A64_xml_00bet7_OPT]

BFM

Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the other bits of the destination register remain unchanged.

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

31	30	29	28	27	26	25	24	23	22	21–16	15–10	9–5	4–0
sf	0	1	1	0	0	1	1	0	N	immr	imms	Rn	Rd
	opc												

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

```
BFM <Wd>, <Wn>, #<immr>, #<imms>
```

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

```
BFM <Xd>, <Xn>, #<immr>, #<imms>
```

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

~~boolean inzero;~~
~~boolean extend;~~
integer R;
~~integer S;~~
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then~~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' UnallocatedEncoding();~~

~~if sf == '1' && N != '1' then ~~ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue~~();~~

~~R = UInt~~();

R =~~(immr);~~
~~S =~~ UInt(immr);
~~(imms);~~
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

Assembler Symbols

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

<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Is preferred when
BFC	`Rn == '11111' && UInt(imms) < UInt(immr)`
BFI	`Rn != '11111' && UInt(imms) < UInt(immr)`
BFXIL	`UInt(imms) >= UInt(immr)`

Operation

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

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

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

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:~~58~~45

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.

31	30	29	28	27	26	25	24	23	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	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
		op														cmode															

16-bit (cmode == 10x1)

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

32-bit (cmode == 0xx1)

```
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;

ImmediateOp operation;
case cmode:op of
    when '0xx01' operation =    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx11' operation =    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x01' operation =    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x11' operation =    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x1' operation =    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '1110x' operation =    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

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

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.
For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	0	1	1	1	0	0	1	1	Rm					0	0	0	1	1	1	Rn					Rd				
								size																							

Three registers of the same type

```
BIC <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 datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean invert = (size<0> == '1');LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AN
```

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

```
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30 29	28 27 26 25 24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0 0	0 1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc			N				

32-bit (sf == 0)

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

64-bit (sf == 1)

```
BIC <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' thenboolean 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 ReservedValue();

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<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;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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	1	0	shift	1	Rm	imm6	Rn	Rd
	opc								N				

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

if sf == '0' && imm6<5> == '1' then~~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~~ ReservedValue();

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<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;
PSTATE.<N,Z,C,V> = result<datasize-1>:case op of
    whenLogicalOp_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.

BIF

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

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

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	1	1	1	Rm	0	0	0	1	1	1	Rn	Rd
								opc2											

Three registers of the same type

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

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

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

Assembler Symbols

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

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

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

```
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];
operand3 = NOT(        operand2 = operand1;
        operand3 = NOT(V[m]);

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

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	1	Rn	Rd
								opc2											

Three registers of the same type

```
BIT <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 datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;VBitOp op;

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

Assembler Symbols

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

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

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

```
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];
operand3 =        operand2 = operand1;
        operand3 = NOT( V[m];[m]);
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);[d] = operand1 EOR ((operand2 EOR operand4)
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to 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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:~~45~~58

BL

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.

31	30	29	28	27	26	25	24	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	1	imm26																									
op																															

26-bit signed PC-relative branch offset

`BL <label>`

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

Assembler Symbols

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

Operation

~~`if branch_type == BranchType_CALL then `~~`X[30] = PC[] + 4;`

`BranchTo(PC[] + offset, BranchType_CALL);`~~`[] + offset, branch_type);`~~

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	0	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	Rn					0	0	0	0	0
							Z		op											A	M						Rm				

Integer

```
BLR <Xn>
```

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

if !pac && m != 0 then
    UnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n !=31 then UnallocatedEncoding();
        n = 30;
        source_is_sp = TRUE;
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
bits(64) target = X[n];[n];
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);

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	Z	0	0	1	1	1	1	1	1	0	0	0	0	1	M	Rn					Rm				
									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>
```

```
integer n = UInt(Rn);
integer m =(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 !if !pac && m != 0 thenUnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET();

if Z == '0' && m != 31 thenthen
        if n !=31 then
    UnallocatedEncoding();();
        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.

Operation

```
bits(64) target = X[n];
bits(64) modifier = if source_is_sp thenif 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);(target, modifier);

if branch_type ==

BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, BranchType_CALL);(target, branch_type);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

BR

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

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

Integer

```
BR <Xn>
```

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

if !pac && m != 0 then
    UnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n !=31 then UnallocatedEncoding();
        n = 30;
        source_is_sp = TRUE;
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
bits(64) target = X[n];[n];
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_CALL then X[30] = PC[] + 4;
BranchTo(target, BranchType_JMP);(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	9-5	4-0
1	1	0	1	0	1	1	Z	0	0	0	1	1	1	1	1	0	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 =(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 !if !pac && m != 0 thenUnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET();

if Z == '0' && m != 31 thenthen
        if n !=31 then
    UnallocatedEncoding();();
        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.

Operation

```
bits(64) target = X[n];
bits(64) modifier = if source_is_sp thenif 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);(target, modifier);

if branch_type ==

BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, BranchType_JMP);(target, branch_type);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

BRK

Breakpoint instruction generates a Breakpoint Instruction exception. The PE records the exception in *ESR_ELx*, using the EC value 0x3c, and captures the value of the immediate argument in *ESR_ELx*.ISS.

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

System

```
BRK #<imm>
```

```
// Empty.bits(16) comment = imm16;
```

Assembler Symbols

<imm>	Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.SoftwareBreakpoint(imm16);(comment);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:~~5845~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	Q	1	0	1	1	1	0	0	1	1	Rm	0	0	0	1	1	1	Rn	Rd
								opc2											

Three registers of the same type

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

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

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

Assembler Symbols

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

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

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

```
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 =[d];
        operand2 = operand1;
        operand3 = NOT( V[d];[m]);
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);[d] = operand1 EOR ((operand2 EOR operand4)
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	1	0	0	0	1	L	1	Rs					o0	1	1	1	1	1	Rn					Rt				
size																															

CASAB (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 UnallocatedEncoding();

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);~~(Rs);~~

~~integer datasize = 8 <<~~

~~UInt(size);~~
~~integer regsize = if datasize == 64 then 64 else 32;~~
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW 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;

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, 1, ldacctype];
[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, 1, stacctype] = newvalue;[address, datasize DIV 8, stacctype] = newvalue;

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- `CASAH` and `CASALH` load from memory with acquire semantics.
- `CASLH` and `CASALH` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	L	1	Rs					o0	1	1	1	1	1	Rn					Rt				
size																															

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

```
CASAH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

CASALH (L == 1 && o0 == 1)

```
CASALH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

CASH (L == 0 && o0 == 0)

```
CASH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

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

```
CASLH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);(Rs);

integer datasize = 8 <<

UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW 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(16) comparevalue;
bits(16) newvalue;
bits(16) data;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

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;[address, datasize DIV 8, stacctype] = newvalue;

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- `CASPA` and `CASPAL` load from memory with acquire semantics.
- `CASPL` and `CASPAL` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is <Ws> and <W(s+1)>, or <Xs> and <X(s+1)>, are restored to the values held in the registers before the instruction was executed.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	sz	0	0	1	0	0	0	0	L	1			Rs			o0	1	1	1	1	1			Rn					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)

```
CASPA <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit CASPAL (sz == 0 && L == 1 && o0 == 1)

```
CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit CASPL (sz == 0 && L == 0 && o0 == 1)

```
CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASP (sz == 1 && L == 0 && o0 == 0)

```
CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASPA (sz == 1 && L == 1 && o0 == 0)

```
CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASPAL (sz == 1 && L == 1 && o0 == 1)

```
CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASPL (sz == 1 && L == 0 && o0 == 1)

```
CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
if Rs<0> == '1' then UnallocatedEncoding();
if Rt<0> == '1' then UnallocatedEncoding();

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 32 << UInt(sz);(sz);
integer regsize = datasize;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW 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.
<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.
<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.
<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.
<X(t+1)>	Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
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 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];
[address, (2 * datasize) DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, (2*datasize) DIV 8, stacctype] = newvalue;
[address, (2 * datasize) DIV 8, stacctype] = newvalue;

if BigEndian() then
    X[s] = ZeroExtend(data<2*datasize-1:datasize>, datasize);(data<2*datasize-1:datasize>, regsize);
    X[s+1] = ZeroExtend(data<datasize-1:0>, datasize);
(data<datasize-1:0>, regsize);
else
    X[s] = ZeroExtend(data<datasize-1:0>, datasize);(data<datasize-1:0>, regsize);
    X[s+1] = ZeroExtend(data<2*datasize-1:datasize>, datasize);(data<2*datasize-1:datasize>, regsize);
```


ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

31	30	29	28	27	26	25	24	23 ... 5	4 ... 0
sf	0	1	1	0	1	0	1	imm19	Rt
							op		

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(operand1) == iszero then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

31	30	29	28	27	26	25	24	23–5	4–0
sf	0	1	1	0	1	0	0	imm19	Rt
							op		

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(operand1) == iszero then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

CCMN (immediate)

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-12	11	10	9-5	4	3-0
sf	0	1	1	1	0	1	0	0	1	0	imm5	cond	1	0	Rn	0	nzcv
	op																

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<imm>	Is a five bit unsigned (positive) immediate encoded in the "imm5" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;
bit carry_in = '0';

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

CCMN (register)

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-12	11	10	9-5	4	3-0
sf	0	1	1	1	0	1	0	0	1	0	Rm	cond	0	0	Rn	0	nzcv
	op																

32-bit (sf == 0)

```
CCMN <Wn>, <Wm>, #<nzcv>, <cond>
```

64-bit (sf == 1)

```
CCMN <Xn>, <Xm>, #<nzcv>, <cond>
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';

if ConditionHolds(cond) then
(condition) 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.

31	30	29	28	27	26	25	24	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	1	1	0	1	0	0	1	0	imm5	cond	1	0	Rn	0	nzcv
	op																

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

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

<imm> Is a five bit unsigned (positive) immediate encoded in the "imm5" field.

<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

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

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2;
bits(datasize) operand2 = imm;
bit carry_in = '0';

if ConditionHolds(cond) then
    operand2 = NOT(imm);
(condition) then
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-12	11	10	9-5	4	3-0
sf	1	1	1	1	0	1	0	0	1	0	Rm	cond	0	0	Rn	0	nzcv
	op																

32-bit (sf == 0)

```
CCMP <Wn>, <Wm>, #<nzcv>, <cond>
```

64-bit (sf == 1)

```
CCMP <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
    operand2 = NOT(operand2);
(condition) then
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

CLS

Count leading sign bits: `Rd = CLS(Rn)`.

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

32-bit (sf == 0)

```
CLS <Wd>, <Wn>
```

64-bit (sf == 1)

```
CLS <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;~~CountOp opcode = if op == '0' then CountOp_CLZ else Cou~~
```

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_CLZ then
    result = CountLeadingZeroBits(operand1);
else
    result = CountLeadingSignBits(operand1);

X[d] = result<datasize-1:0>;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

CLZ

Count leading zero bits: `Rd = CLZ(Rn)`.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9-5	4-0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	Rn	Rd
																					op		

32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;~~CountOp opcode = if op == '0' then CountOp_CLZ else Cou~~
```

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_CLZ then~~
~~    result =~~ CountLeadingZeroBits~~(operand1);~~
~~else~~
~~    result = CountLeadingSignBits~~(operand1);
X[d] = result<datasize-1:0>;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11-10	9-5	4-0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	1	0	0	sz	Rn	Rd
															C			

CRC32B (sf == 0 && sz == 00)

```
CRC32B <Wd>, <Wn>, <Wm>
```

CRC32H (sf == 0 && sz == 01)

```
CRC32H <Wd>, <Wn>, <Wm>
```

CRC32W (sf == 0 && sz == 10)

```
CRC32W <Wd>, <Wn>, <Wm>
```

CRC32X (sf == 1 && sz == 11)

```
CRC32X <Wd>, <Wn>, <Xm>
```

```
if !HaveCRCExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding();
integer size = 8 << UInt(sz);(sz);    // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
<Wm>	Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	20-16	15	14	13	12	11-10	9-5	4-0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	1	0	1	sz	Rn	Rd
															C			

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

```
CRC32CB <Wd>, <Wn>, <Wm>
```

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

```
CRC32CH <Wd>, <Wn>, <Wm>
```

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

```
CRC32CW <Wd>, <Wn>, <Wm>
```

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

```
CRC32CX <Wd>, <Wn>, <Xm>
```

```
if !HaveCRCExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding();
integer size = 8 << UInt(sz);(sz);   // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
<Wm>	Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE prediction state for any SVE instructions.

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

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

System

```
CSDB
```

```
~~SystemHintOp op;~~

~~case CRm:op2 of~~
~~    when '0000 000' op = SystemHintOp_NOP;~~
~~    when '0000 001' op = SystemHintOp_YIELD;~~
~~    when '0000 010' op = SystemHintOp_WFE;~~
~~    when '0000 011' op = SystemHintOp_WFI;~~
~~    when '0000 100' op = SystemHintOp_SEV;~~
~~    when '0000 101' op = SystemHintOp_SEVL;~~
~~    when '0000 111'~~
~~        SEE "XPACLRI";~~
~~    when '0001 xxx'~~
~~        SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";~~
~~    when '0010 000'~~
~~        if !HaveRASExt() then EndOfInstruction();                // 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";~~
~~    otherwise  EndOfInstruction~~// Empty.~~();                                  // Instruction executes as~~
```

Operation

```
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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();();

    otherwise // do nothing
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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
sf	0	0	1	1	0	1	0	1	0	0	Rm					cond				0	0	Rn					Rd				
	op																				o2										

32-bit (sf == 0)

```
CSEL <Wd>, <Wn>, <Wm>, <cond>
```

64-bit (sf == 1)

```
CSEL <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;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.

Operation

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

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

31	30	29	28	27	26	25	24	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	0	Rm	cond	0	1	Rn	Rd
	op													o2		

32-bit (sf == 0)

```
CSINC <Wd>, <Wn>, <Wm>, <cond>
```

64-bit (sf == 1)

```
CSINC <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;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

Alias	Is preferred when
CINC	`Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm`
CSET	`Rm == '11111' && cond != '111x' && Rn == '11111'`

Operation

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

if ConditionHolds(cond) then
(condition) 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-12	11	10	9-5	4-0
sf	1	0	1	1	0	1	0	1	0	0	Rm	cond	0	0	Rn	Rd
	op													o2		

32-bit (sf == 0)

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

64-bit (sf == 1)

```
CSINV <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;~~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

Alias	Is preferred when
CINV	`Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm`
CSETM	`Rm == '11111' && cond != '111x' && Rn == '11111'`

Operation

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

if ConditionHolds(cond) then
(condition) then
    result = operand1;
else
    result = NOT(operand2);    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	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	Rm					cond				0	1	Rn					Rd				
	op																				o2										

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;~~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

Alias	Is preferred when
CNEG	`cond != '111x' && Rn == Rm`

Operation

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

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

DCPS1

Debug Change PE State to EL1, when executed in Debug state:

- If executed at EL0 changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS1 instruction is:

- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

- *ELR_ELx* becomes UNKNOWN.
- *SPSR_ELx* becomes UNKNOWN.
- *ESR_ELx* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_ELx*.EE.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and *HCR_EL2*.TGE == 1.

This instruction is always UNDEFINED in Non-debug state.

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

31	30	29	28	27	26	25	24	23	22	21	20–5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16	0	0	0	0	1
															LL	

System

```
DCPS1 {#<imm>}
```

```
if !bits(2) target_level = LL;
if LL == '00' thenUnallocatedEncoding();
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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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 in Secure state if EL2 is 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	20-5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16	0	0	0	1	0
															LL	

System

```
DCPS2 {#<imm>}
```

```
if !bits(2) target_level = LL;
if LL == '00' thenUnallocatedEncoding();
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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	20–5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16	0	0	0	1	1
															LL	

System

```
DCPS3 {#<imm>}
```

```
if !~~bits(2) target_level = LL;~~
~~if LL == '00' then UnallocatedEncoding();~~
~~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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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

System

```
DMB <option>|#<imm>
```

```
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;
    otherwise UnallocatedEncoding;
();

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> == '00' then
            op = MemBarrierOp_SSBB;
        elsif CRm<3:2> == '01' then
            op = MemBarrierOp_PSSBB;
        else
            types = MBReqTypes_All;
            domain = MBReqDomain_FullSystem;
```

Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

SY

Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST

Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD

Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 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

```
case op of
    when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);
    when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
    when MemBarrierOp_ISBInstructionSynchronizationBarrier();
    when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
    when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization 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	1	0	0	0	0	0	0	1	1	0	0	1	1	!= 0x00				1	0	0	1	1	1	1	1
																				CRm					opc						

System

```
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 UnallocatedEncoding();

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> == '00' then
            op = MemBarrierOp_SSBB;
        elsif CRm<3:2> == '01' then
            op = MemBarrierOp_PSSBB;
        else
            types = MBReqTypes_All;
            domain = MBReqDomain_FullSystem;
```

Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

SY

Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST

Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD

Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 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

```
case op of
    when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);
    when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
    when MemBarrierOp_ISBInstructionSynchronizationBarrier();
    when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
    when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

31	30	29	28	27	26	25	24	23-22	21	20-16	15-10	9-5	4-0
sf	1	0	0	1	0	1	0	shift	1	Rm	imm6	Rn	Rd
	opc								N				

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<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 EOR 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) **htmldiff from-ISA_v84A_A64_xml_00bet7** **(new) ISA_v84A_A64_xml_00bet7_OPT**

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.

31	30	29	28	27	26	25	24	23	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	1	Rm					0	0	0	1	1	1	Rn					Rd				
								opc2																							

Three registers of the same type

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

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

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

Assembler Symbols

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

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

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

```
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	21-16	15-10	9-5	4-0
sf	1	0	1	0	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>
```

```
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' thenboolean 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 ReservedValue();
(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 thencase 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	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	0	shift	0	Rm	imm6	Rn	Rd
	opc								N				

32-bit (sf == 0)

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

64-bit (sf == 1)

```
EOR <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~~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~~ ReservedValue();

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<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);

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores *PSTATE* from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERET is UNDEFINED at EL0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
																				A	M			Rn					op4		

System

```
ERET
```

```
if PSTATE.EL == EL0 then UnallocatedEncoding();();
boolean pac = (A == '1');
boolean use_key_a = (M == '0');

if !pac && op4 != '00000' thenUnallocatedEncoding();
elsif pac && (!HavePACExt() || op4 != '11111') then
    UnallocatedEncoding();

if Rn != '11111' then
    UnallocatedEncoding();
```

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

AArch64.ExceptionReturn(target, SPSR[]);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1
																				A				Rn					op4		

ERETAA (M == 0)

```
ERETAA
```

ERETAB (M == 1)

```
ERETAB
```

```
if PSTATE.EL == EL0 then UnallocatedEncoding();
~~boolean pac = (A == '1');~~
boolean use_key_a = (M == '0');

if !~~if !pac && op4 != '00000' thenUnallocatedEncoding();~~
~~elsif pac && (!~~HavePACExt~~() || op4 != '11111') then~~
    ~~UnallocatedEncoding~~() then~~();~~

~~if Rn != '11111' then~~
    UnallocatedEncoding();
```

Operation

```
AArch64.CheckForERetTrap(TRUE, use_key_a);
bits(64) target;

if use_key_a then
    target =~~(pac, use_key_a);~~
~~bits(64) target = ELR[];~~

~~if pac then~~
~~    if use_key_a then~~
~~        target = ~~AuthIA(ELR[], SP[]);
else
    target = AuthIB(ELR[], SP[]);

AArch64.ExceptionReturn(target, SPSR[]);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1	1	1	1	1
																				CRm				op2							

System

```
ESB
```

```
SystemHintOpif !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() then;
    when '0011 xxx'
        SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    otherwise EndOfInstruction();();                                  // Instruction executes as NOP
```

Operation

```
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();
TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();();

    otherwise // do nothing
```

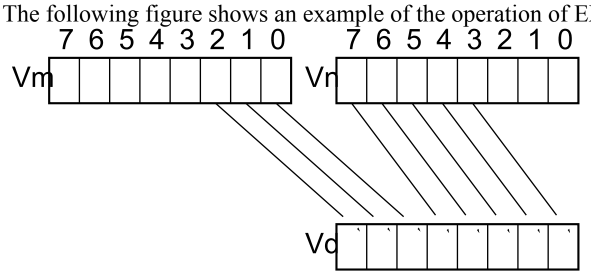
Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23	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	Rm					0	imm4				0	Rn					Rd				

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 UnallocatedEncoding();

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”:

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.

<index> Is the lowest numbered byte element to be extracted, encoded in “Q:imm4”:

Q	imm4<3>	<index>
0	0	imm4<2:0>
0	1	RESERVED
1	x	imm4

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23	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	0	0	0	1	1	0	0	0	0	Rn					Rd				
															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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
```

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	type	1	Rm	0	0	1	0	1	0	Rn	Rd
														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>
```

```
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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

~~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

```
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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	9-5	4-0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	1	1	0	Rn	Rd
									sz														

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = 32;if sz == '1' thenReservedValue();
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = ReduceOp_FADD;
```

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	9-5	4-0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	1	1	0	Rn	Rd

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

ReduceOp op = ReduceOp_FADD;
```

Assembler Symbols

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

sz	<V>
0	H
1	RESERVED

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

sz	<V>
0	S
1	D

<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”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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
0	Q	1	0	1	1	1	0	size		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 UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then ReservedValue();
if Q == '0' && size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then ReservedValue();
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”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<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”:

rot	<rotate>
0	90
1	270

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
    case rot of
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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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 in the *CPACR_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	20-16	15-12	11	10	9-5	4	3-0
0	0	0	1	1	1	1	0	type	1	Rm	cond	0	1	Rn	0	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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

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

NaNs

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

```
CheckFPAdvSIMDEnabled64();

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

operand2 = V[m];

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

FCCMPE

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*.{N, Z, C, V} flags. If the condition does not pass then the *PSTATE*.{N, Z, C, V} flags are set to the flag bit specifier.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

31	30	29	28	27	26	25	24	23-22	21	20-16	15-12	11	10	9-5	4	3-0
0	0	0	1	1	1	1	0	type	1	Rm	cond	0	1	Rn	1	nzcv
															op	

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>
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);

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

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

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

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

FCCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation

```
CheckFPAdvSIMDEnabled64();

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

operand2 = V[m];

if ConditionHolds(cond) then
(condition) then
    flags = FPCompare(operand1, operand2, TRUE, FPCR);
(operand1, operand2, signal_all_nans, FPCR);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

FCMLA (by element)

Floating-point Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

Vector
(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				0	rot		1	H	0	Rn					Rd				

(size == 01)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

(size == 10)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

```
if !HaveFCADDExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
if size == '00' || size == '11' then ReservedValue();
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 ReservedValue();
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then ReservedValue();
if size == '01' && H == '1' && Q == '0' then();
if size == '01' && H == '1' && Q=='0' then ReservedValue();
```

Assembler Symbols

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

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

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

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

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

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:H:L”:

size	<index>
00	RESERVED
01	H:L
10	H
11	RESERVED

<rotate> Is the rotation, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

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]);
[operand2, index*2,esize]); 
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, index*2+1, esize]);
            element4 = Elem[operand1, e*2, 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;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	Rn					Rd				

Three registers of the same type

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

```
if !HaveFCADDExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then ReservedValue();
if Q == '0' && size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then ReservedValue();
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”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<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”:

rot	<rotate>
00	0
01	90
10	180
11	270

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, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, e*2+1, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, e*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	0	1	1	1	1	0	type		1	Rm					cond				1	1	Rn					Rd				

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

```
FCSEL <Hd>, <Hn>, <Hm>, <cond>
```

Single-precision (type == 00)

```
FCSEL <Sd>, <Sn>, <Sm>, <cond>
```

Double-precision (type == 01)

```
FCSEL <Dd>, <Dn>, <Dm>, <cond>
```

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

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

bits(4) condition = cond;
```

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

Operation

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

result = if ConditionHolds(cond) then(condition) then V[n] else V[m];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:~~58~~45

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	type		1	0	0	1	0	0	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTAS <Wd>, <Hn>
```

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

```
FCVTAS <Xd>, <Hn>
```

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

```
FCVTAS <Wd>, <Sn>
```

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

```
FCVTAS <Xd>, <Sn>
```

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

```
FCVTAS <Wd>, <Dn>
```

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

```
FCVTAS <Xd>, <Dn>
```

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

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

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'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 UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS() then
            fltsize = 16;
        else;
    otherwise
            UnallocatedEncoding();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

fltval =case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR,(fltval, 0, unsigned, FPCR, rounding); FPRounding_TIEAWAYX);[
    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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	type	1	0 0	1 0 1	0	0	0	0	0	0	Rn	Rd
										rmode	opcode								

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTAU <Wd>, <Hn>
```

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

```
FCVTAU <Xd>, <Hn>
```

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

```
FCVTAU <Wd>, <Sn>
```

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

```
FCVTAU <Xd>, <Sn>
```

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

```
FCVTAU <Wd>, <Dn>
```

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

```
FCVTAU <Xd>, <Dn>
```

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

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

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'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 UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS() then
            fltsize = 16;
        else;
    otherwise
            UnallocatedEncoding();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

fltval =case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR,(fltval, 0, unsigned, FPCR, rounding); FPRounding_TIEAWAYX);[c
    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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

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

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTMS <Wd>, <Hn>
```

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

```
FCVTMS <Xd>, <Hn>
```

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

```
FCVTMS <Wd>, <Sn>
```

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

```
FCVTMS <Xd>, <Sn>
```

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

```
FCVTMS <Wd>, <Dn>
```

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

```
FCVTMS <Xd>, <Dn>
```

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'    when '10'
        if opcode<2:1>:rmode != '11 01' then
        UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding(rmode);();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

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

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'    when '10'
        if opcode<2:1>:rmode != '11 01' then
        UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding(rmode);();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

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;(intval<31:0>, 64);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) **htmldiff from-ISA_v84A_A64_xml_00bet7** **(new) ISA_v84A_A64_xml_00bet7_OPT**

FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

31	30	29	28	27	26	25	24	23-22	21	20-19	18-16	15-10	9-5	4-0
sf	0	0	1	1	1	1	0	type	1	0 0	0 0 0	0 0 0 0 0 0	Rn	Rd
										rmode	opcode			

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTNS <Wd>, <Hn>
```

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

```
FCVTNS <Xd>, <Hn>
```

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

```
FCVTNS <Wd>, <Sn>
```

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

```
FCVTNS <Xd>, <Sn>
```

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

```
FCVTNS <Wd>, <Dn>
```

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

```
FCVTNS <Xd>, <Dn>
```

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'    when '10'
        if opcode<2:1>:rmode != '11 01' then
        UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding(rmode);();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

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

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTNU <Wd>, <Hn>
```

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

```
FCVTNU <Xd>, <Hn>
```

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

```
FCVTNU <Wd>, <Sn>
```

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

```
FCVTNU <Xd>, <Sn>
```

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

```
FCVTNU <Wd>, <Dn>
```

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

```
FCVTNU <Xd>, <Dn>
```

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'    when '10'
        if opcode<2:1>:rmode != '11 01' then
        UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding(rmode);();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

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;(intval<31:0>, 64);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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.

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

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTPS <Wd>, <Hn>
```

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

```
FCVTPS <Xd>, <Hn>
```

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

```
FCVTPS <Wd>, <Sn>
```

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

```
FCVTPS <Xd>, <Sn>
```

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

```
FCVTPS <Wd>, <Dn>
```

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

```
FCVTPS <Xd>, <Dn>
```

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'    when '10'
        if opcode<2:1>:rmode != '11 01' then
        UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding(rmode);();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	type	1	0 1	0 0 1	0 0 0 0 0 0	Rn	Rd
										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'~~    when '10'~~
~~        if opcode<2:1>:rmode != '11 01' then~~
        UnallocatedEncoding();
~~        fltsize = 128;~~
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();~~
~~        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 UnallocatedEncoding();~~
~~        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 UnallocatedEncoding();~~
~~        rounding = FPRounding_ZERO;~~
~~        unsigned = (opcode<0> == '1');~~
~~        op = FPConvOp_CVT_FtoI_JS;~~
~~    otherwise~~
~~        UnallocatedEncoding~~(rmode);~~();~~
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

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;(intval<31:0>, 64);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	1	1	1	0	type	0	1	1	0	0	0	scale	Rn	Rd
										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'integer fltsize; FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
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
        UnallocatedEncoding(scale);();
```

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

```
CheckFPAdvSIMDEnabled64();

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

fltval =case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
intval = FPToFixed(fltval, fracbits, FALSE, FPCR,(fltval, fracbits, unsigned, FPCR, rounding); FPRoundi
    when
FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
        V[d] = intval;[d] = fltval;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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.

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

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTZS <Wd>, <Hn>
```

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

```
FCVTZS <Xd>, <Hn>
```

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

```
FCVTZS <Wd>, <Sn>
```

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

```
FCVTZS <Xd>, <Sn>
```

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

```
FCVTZS <Wd>, <Dn>
```

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

```
FCVTZS <Xd>, <Dn>
```

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'    when '10'
        if opcode<2:1>:rmode != '11 01' then
        UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding(rmode);();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	1	0	type	0	1	1	0	0	1	scale	Rn	Rd
										rmode		opcode					

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'integer fltsize; FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
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
        UnallocatedEncoding(scale);();
```

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

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

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.

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

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'    when '10'
        if opcode<2:1>:rmode != '11 01' then
        UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding(rmode);();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

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

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;(intval<31:0>, 64);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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)

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

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

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS() then;
    otherwise UnallocatedEncoding();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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 *CPACR_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	20–16	15	14–10	9–5	4–0
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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();();

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

```
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;
```

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	1	0	type	1	Rm	0	1	0	0	1	0	Rn	Rd
													op					

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

```
FMAX <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FMAX <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FMAX <Dd>, <Dn>, <Dm>
```

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

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();FPMaxMinOp operation;
case op of
    when '00' operation = FPMaxMinOp_MAX;
    when '01' operation = FPMaxMinOp_MIN;
    when '10' operation = FPMaxMinOp_MAXNUM;
    when '11' operation = FPMaxMinOp_MINNUM;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

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

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

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

result =case operation of
    when FPMaxMinOp_MAX    result = FPMax(operand1, operand2, FPCR);
    when FPMaxMinOp_MIN    result = FPMin(operand1, operand2, FPCR);
    when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
    when FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

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

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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();FPMaxMinOp operation;
case op of
    when '00' operation = FPMaxMinOp_MAX;
    when '01' operation = FPMaxMinOp_MIN;
    when '10' operation = FPMaxMinOp_MAXNUM;
    when '11' operation = FPMaxMinOp_MINNUM;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

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

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

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

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

result =case operation of
    when FPMaxMinOp_MAX    result = FPMax(operand1, operand2, FPCR);
    when FPMaxMinOp_MIN    result = FPMin(operand1, operand2, FPCR);
    when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
    when FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	9-5	4-0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn	Rd
								o1	sz														

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = 32;if sz == '1' thenReservedValue();
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	9-5	4-0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn	Rd
								o1															

Single-precision and double-precision

```
FMAXNMP <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

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

sz	<V>
0	H
1	RESERVED

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

sz	<V>
0	S
1	D

<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”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1																							

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FM
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

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

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

if sz:Q != '01' then ReservedValue();    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FM
```

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”:

sz	<V>
0	S
1	RESERVED

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

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

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1	sz																						

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = 32;~~if sz == '1' thenReservedValue();~~
~~integer datasize = esize * 2;~~
~~integer elements = 2;~~

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

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

```
FMAXP <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_FMIN else ReduceOp_FMAX;~~
```

Assembler Symbols

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

sz	<V>
0	H
1	RESERVED

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

sz	<V>
0	S
1	D

<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”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	9-5	4-0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn	Rd
								o1															

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;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	9-5	4-0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	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 ReservedValue();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;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”:

sz	<V>
0	S
1	RESERVED

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

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

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) **htmldiff from-ISA_v84A_A64_xml_00bet7** **(new) ISA_v84A_A64_xml_00bet7_OPT**

FMIN (scalar)

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

31	30	29	28	27	26	25	24	23-22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	0	0	1	1	1	1	0	type	1	Rm	0	1	0	1	1	0	Rn	Rd
													op					

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

```
FMIN <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FMIN <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FMIN <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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();FPMaxMinOp operation;
case op of
    when '00' operation = FPMaxMinOp_MAX;
    when '01' operation = FPMaxMinOp_MIN;
    when '10' operation = FPMaxMinOp_MAXNUM;
    when '11' operation = FPMaxMinOp_MINNUM;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

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

Operation

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

result =case operation of
    when FPMaxMinOp_MAX    result = FPMax(operand1, operand2, FPCR);
    when FPMaxMinOp_MIN    result = FPMin(operand1, operand2, FPCR);
    when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);
    when FPMaxMinOp_MINNUM result = FPMinNum(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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 *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.

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

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>
```

```
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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();FPMaxMinOp operation;
case op of
    when '00' operation = FPMaxMinOp_MAX;
    when '01' operation = FPMaxMinOp_MIN;
    when '10' operation = FPMaxMinOp_MAXNUM;
    when '11' operation = FPMaxMinOp_MINNUM;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

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

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

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

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

result =~~case operation of~~
~~    when FPMaxMinOp_MAX    result = FPMax(operand1, operand2, FPCR);~~
~~    when FPMaxMinOp_MIN    result = FPMin(operand1, operand2, FPCR);~~
~~    when FPMaxMinOp_MAXNUM result = FPMaxNum(operand1, operand2, FPCR);~~
~~    when FPMaxMinOp_MINNUM result = ~~FPMinNum(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1	sz																						

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = 32;~~if sz == '1' thenReservedValue();~~
~~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	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1																							

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

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

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<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":

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

sz	<T>
0	2S
1	2D

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn	Rd
								o1															

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;~~integer datasize = if Q == '1' then 128 else 64;~~
~~integer elements = datasize DIV esize;ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FM~~
```

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	9-5	4-0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn	Rd
								o1															

Single-precision and double-precision

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

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

if sz:Q != '01' then ReservedValue();    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;~~integer datasize = if Q == '1' then 128 else 64;~~
~~integer elements = datasize DIV esize;ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FM~~
```

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”:

sz	<V>
0	S
1	RESERVED

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

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

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1	sz																						

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = 32;~~if sz == '1' thenReservedValue();~~
~~integer datasize = esize * 2;~~
~~integer elements = 2;~~

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

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

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

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

integer esize = 32;
integer datasize = 64;~~integer esize = 32 << UInt(sz);~~
~~integer datasize = esize * 2;~~
~~integer elements = 2;~~

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

Assembler Symbols

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

sz	<V>
0	H
1	RESERVED

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

sz	<V>
0	S
1	D

<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”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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 UnallocatedEncoding();

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

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;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 ReservedValue();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;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”:

sz	<V>
0	S
1	RESERVED

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

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

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	19-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	1	1	sz	L	M	Rm	0	0	0	0	H	0	Rn	Rd
														S						

FMLAL

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

```
if !HaveFP16MulNoRoundingToFP32Ext() then UnallocatedEncoding();
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 ReservedValue();
integer index = UInt(H:L:M);
(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
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)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15	14	13	12	11	10	9-5	4-0
0	Q	1	0	1	1	1	1	1	sz	L	M	Rm	1	0	0	0	H	0	Rn	Rd
														S						

FMLAL2

```
FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

```
if !HaveFP16MulNoRoundingToFP32Ext() then UnallocatedEncoding();
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 ReservedValue();
integer index = UInt(H:L:M);
(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
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”:

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	2H
1	4H

<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];
[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;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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)

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

FMLAL

```
FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
if !HaveFP16MulNoRoundingToFP32Ext() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then ReservedValue();
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)

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

FMLAL2

```
FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
if !HaveFP16MulNoRoundingToFP32Ext() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then ReservedValue();
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”:

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	2H
1	4H

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
[n,part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
[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([result,e,esize] = FPMulAddH(Elem[operand3, e, esize], element1,
V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	1	sz	L	M	Rm				0	1	0	0	H	0	Rn					Rd				
																	S														

FMLSL

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

```
if !HaveFP16MulNoRoundingToFP32Ext() then UnallocatedEncoding();
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 ReservedValue();
integer index = UInt(H:L:M);
(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
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
0	Q	1	0	1	1	1	1	1	sz	L	M	Rm				1	1	0	0	H	0	Rn					Rd				
																	S														

FMLSL2

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

```
if !HaveFP16MulNoRoundingToFP32Ext() then UnallocatedEncoding();
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 ReservedValue();
integer index = UInt(H:L:M);
(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
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”:

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	2H
1	4H

<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];
[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;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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

FMLSL

```
FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
if !HaveFP16MulNoRoundingToFP32Ext() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then ReservedValue();
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
0	Q	1	0	1	1	1	0	1	sz	1	Rm					1	1	0	0	1	1	Rn					Rd				
								S																							

FMLSL2

```
FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
if !HaveFP16MulNoRoundingToFP32Ext() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then ReservedValue();
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":

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	2H
1	4H

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
[n,part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
[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([result,e,esize] = FPMulAddH(Elem[operand3, e, esize], element1,
V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

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

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	1	1	d	e	f	g	h	Rd				

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer rd = UInt(Rd);

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

imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:(imm8<6>,2):imm8<5:0>:Zeros(6);

imm = Replicate(imm16, datasize DIV 16);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	0	1	d	e	f	g	h	Rd				
																cmode															

Single-precision (op == 0)

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

Double-precision (Q == 1 && op == 1)

```
FMOV <Vd>.2D, #<imm>
```

```
integer rd = UInt(Rd);

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

if cmode:op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' thenbits(64) imm64; ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        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”:

Q	<T>
0	4H
1	8H

For the single-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in A64 floating-point instructions*.

Operation

```
CheckFPAdvSIMDEnabled64();

V[rd] = imm;
```

FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

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

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

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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();~~FPUnaryOp fpop;~~
~~case opc of~~
~~    when '00' fpop = FPUnaryOp_MOV;~~
~~    when '01' fpop = FPUnaryOp_ABS;~~
~~    when '10' fpop = FPUnaryOp_NEG;~~
~~    when '11' fpop = FPUnaryOp_SQRT;~~
```

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];[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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

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

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

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 UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
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];
[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;
[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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23 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	1	Ra	Rn	Rd
									o1		o0			

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>
```

```
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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();();

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

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

operand1 =if opa_neg then operanda = FPNeg(operand1);
result =(operanda);
if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

FMUL (scalar)

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

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

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

```
FMUL <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FMUL <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FMUL <Dd>, <Dn>, <Dm>
```

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

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

boolean negated = (op == '1');
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

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

result = FPMul(operand1, operand2, FPCR);~~(operand1, operand2, FPCR);~~

~~if negated then result =~~

~~FPNeg(result);~~

V[d] = result;
```

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.

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

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

```
FNEG <Hd>, <Hn>
```

Single-precision (type == 00)

```
FNEG <Sd>, <Sn>
```

Double-precision (type == 01)

```
FNEG <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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();~~FPUnaryOp fpop;~~
~~case opc of~~
~~    when '00' fpop = FPUnaryOp_MOV;~~
~~    when '01' fpop = FPUnaryOp_ABS;~~
~~    when '10' fpop = FPUnaryOp_NEG;~~
~~    when '11' fpop = FPUnaryOp_SQRT;~~
```

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

31	30	29	28	27	26	25	24	23-22	21	20-16	15	14-10	9-5	4-0
0	0	0	1	1	1	1	1	type	1	Rm	0	Ra	Rn	Rd
									o1		o0			

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

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

Single-precision (type == 00)

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

Double-precision (type == 01)

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

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

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();~~();~~

~~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

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

operanda =if opa_neg then operanda = FPNeg(operanda);
operand1 =if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23-22	21	20-16	15	14-10	9-5	4-0
0	0	0	1	1	1	1	1	type	1	Rm	1	Ra	Rn	Rd
									o1		o0			

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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();();

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

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

operanda =if opa_neg then operanda = FPNeg(operanda);
result =if op1_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

31	30	29	28	27	26	25	24	23-22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	0	0	1	1	1	1	0	type	1	Rm	1	0	0	0	1	0	Rn	Rd
											op							

Half-precision (type == 11)

(ARMv8.2)

```
FNMUL <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FNMUL <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FNMUL <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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();();
```

~~boolean negated = (op == '1');~~

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

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

result = FPMul(operand1, operand2, FPCR);

result =~~if negated then result =~~ FPNeg(result);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	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>
```

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

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

boolean exact = FALSE;FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    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,(operand, FPCR, rounding, exact); FPRounding_TIEAWAY, FALSE);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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 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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();();

boolean exact = FALSE;

FPRounding rounding;
rounding =case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    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, FALSE);(operand, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

31	30	29	28	27	26	25	24	23 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 0	1 0 0 0 0	Rn	Rd
											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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();();

boolean exact = FALSE;

FPRounding rounding;
rounding =case rmode of
    when '0xx' rounding = FPDecodeRounding('10');(rmode<1:0>);
    when '100' rounding =FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    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, FALSE);~~(operand, FPCR, rounding, exact);~~

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

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

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

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

Single-precision (type == 00)

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

Double-precision (type == 01)

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

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

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

boolean exact = FALSE;

FPRounding rounding;
rounding =case rmode of
    when '0xx' rounding = FPDecodeRounding('00');(rmode<1:0>);
    when '100' rounding =FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    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, FALSE);(operand, FPCR, rounding, exact);

V[d] = result;
```


ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	0	1	1	1	1	0	type		1	0	0	1	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 datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();();

boolean exact = FALSE;

FPRounding rounding;
rounding =case rmode of
    when '0xx' rounding = FPDecodeRounding('01');(rmode<1:0>);
    when '100' rounding =FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    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, FALSE);(operand, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

An Inexact exception is raised when the result value is not numerically equal to the input value. 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	0	1	0	0	0	0	Rn					Rd				
															rmode																

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

```
FRINTX <Hd>, <Hn>
```

Single-precision (type == 00)

```
FRINTX <Sd>, <Sn>
```

Double-precision (type == 01)

```
FRINTX <Dd>, <Dn>
```

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

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

boolean exact = FALSE;

FPRounding rounding;
rounding =case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    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);(operand, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();();

boolean exact = FALSE;

FPRounding rounding;
rounding =case rmode of
    when '0xx' rounding = FPDecodeRounding('11');(rmode<1:0>);
    when '100' rounding =FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    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, FALSE);(operand, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

31	30	29	28	27	26	25	24	23-22	21	20	19	18	17	16	15	14	13	12	11	10	9-5	4-0
0	0	0	1	1	1	1	0	type	1	0	0	0	0	1	1	1	0	0	0	0	Rn	Rd
														opc								

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

```
FSQRT <Hd>, <Hn>
```

Single-precision (type == 00)

```
FSQRT <Sd>, <Sn>
```

Double-precision (type == 01)

```
FSQRT <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' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
```

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 =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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

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

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

```
FSUB <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FSUB <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FSUB <Dd>, <Dn>, <Dm>
```

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

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

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

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

result =if sub_op then
    result = FPSub(operand1, operand2, FPCR);(operand1, operand2, FPCR);
else
    result =
FPAdd(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

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

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_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";
```
`    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.

Operation

```
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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

    otherwise    // do nothing
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	20-5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16	0	0	0	1	0

System

```
HVC #<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
    UnallocatedEncoding();

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);~~
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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).

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	1	0	0	1	1	1	0	0	0	0	imm5	0	0	0	1	1	1	Rn	Rd

Advanced SIMD

```
INS <Vd>.<Ts>[<index>], <R><n>
```

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

integer size = LowestSetBit(imm5);

if size > 3 then UnallocatedEncoding();
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”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

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

Operation

```
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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

System

```
ISB {<option>|#<imm>}
```

```
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;
    otherwise UnallocatedEncoding;
();

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> == '00' then
            op = MemBarrierOp_SSBB;
        elsif CRm<3:2> == '01' then
            op = MemBarrierOp_PSSBB;
        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.

Operation

```
case op of
    when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);
    when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
    when MemBarrierOp_ISBInstructionSynchronizationBarrier();
    when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
    when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
```

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	1	0	0	0	0	0	0	x	x	1	x	size		Rn					Rt				
									L							opcode															

One register (opcode == 0111)

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

Two registers (opcode == 1010)

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

Three registers (opcode == 0110)

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

Four registers (opcode == 0010)

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

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

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm					x	x	1	x	size		Rn					Rt				
									L							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”:

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

<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”:

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<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 = 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 UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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	0	1	1	0	1	0	1	0	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	1	0	Rm					x	x	0	S	size		Rn					Rt				
									L	R							opcode														

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);     // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);     // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);     // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);     // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

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

No offset

```
LD1R { <Vt>.<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	1	1	1	0	Rm	1	1	0	0	size	Rn	Rt
									L	R			opcode		S			

Immediate offset (Rm == 11111)

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

Register offset (Rm != 11111)

```
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”:

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

<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”:

size	<imm>
00	#1
01	#2
10	#4
11	#8

<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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	size		Rn					Rt				
									L							opcode															

No offset

```
LD2 { <Vt>.<T>, <Vt2>.<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	Rm					1	0	0	0	size		Rn					Rt				
									L							opcode															

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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<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”:

Q	<imm>
0	#16
1	#32

<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 = 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 UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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
0	Q	0	0	1	1	0	1	0	1	1	0	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
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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

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

```
LD2R { <Vt>.<T>, <Vt2>.<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	1	1	1	1	Rm					1	1	0	0	size		Rn					Rt				
									L	R							opcode		S												

Immediate offset (Rm == 11111)

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

Register offset (Rm != 11111)

```
LD2R { <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”:

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

<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 “size”:

size	<imm>
00	#2
01	#4
10	#8
11	#16

<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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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;
```

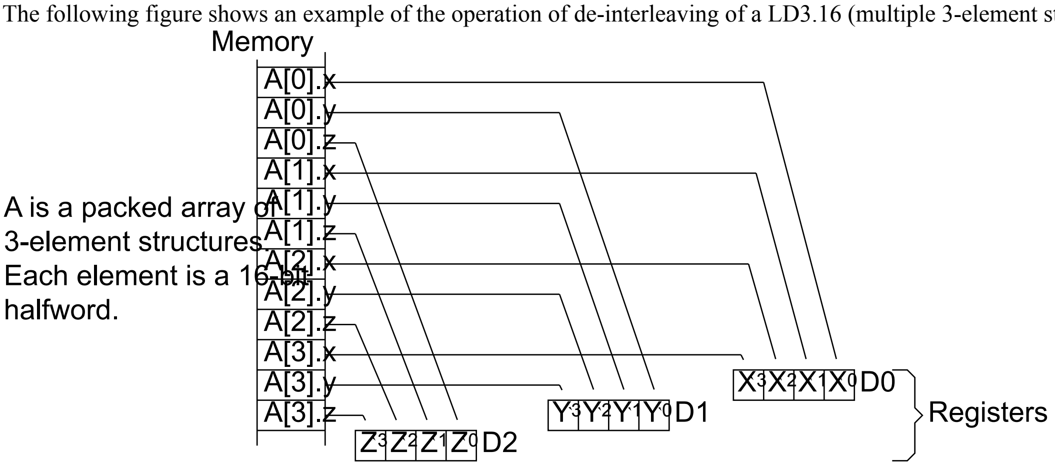
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

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:.



Depending on the settings in the *CPACR_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	1	0	0	0	0	0	0	0	1	0	0	size		Rn					Rt				
									L								opcode														

No offset

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

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

Post-index

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

Immediate offset (Rm == 11111)

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

Register offset (Rm != 11111)

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

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

Assembler Symbols

<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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<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”:

Q	<imm>
0	#24
1	#48

<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 = 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 UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

LD3 (single structure)

Load single 3-element structure to one lane of three registers). This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	1	S	size		Rn					Rt				
									L	R							opcode														

8-bit (opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

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

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

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

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

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

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					x	x	1	S	size		Rn					Rt				
									L	R							opcode														

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

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

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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

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

31	30	29	28	27	26	25	24	23	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	0	Rm					1	1	1	0	size		Rn					Rt				
									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>
```

```
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”:

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

<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”:

size	<imm>
00	#3
01	#6
10	#12
11	#24

<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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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

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

No offset

```
LD4 { <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	Rm	0	0	0	0	size	Rn	Rt
									L			opcode						

Immediate offset (Rm == 11111)

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

Register offset (Rm != 11111)

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

```
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<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”:

Q	<imm>
0	#32
1	#64

<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 = 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 UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	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)

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

16-bit (opcode == 011 && size == x0)

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

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

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

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

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.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	1	1	Rm	x	x	1	S	size	Rn	Rt
									L	R			opcode					

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.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.
<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.

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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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

No offset

```
LD4R { <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	1	1	1	1	Rm	1	1	1	0	size	Rn	Rt
									L	R			opcode		S			

Immediate offset (Rm == 11111)

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

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”:

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

<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 “size”:

size	<imm>
00	#4
01	#8
10	#16
11	#32

<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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);     // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);     // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);     // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);     // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	0	0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size												opc				

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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STADD, STADDL	`A == '0' && Rt == '11111'`

Operation

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

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

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- LDADDB has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STADDB, STADDLB.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	0	0	0	0	0	Rn	Rt
size													opc						

LDADDAB (A == 1 && R == 0)

```
LDADDAB <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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STADDB, STADDLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

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
    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);(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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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 STADDH, STADDLH.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	0	0	0	0	0	Rn	Rt
size													opc						

LDADDAH (A == 1 && R == 0)

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDALH (A == 1 && R == 1)

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDH (A == 0 && R == 0)

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDLH (A == 0 && R == 1)

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STADDH, STADDLH	`A == '0' && Rt == '11111'`

Operation

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

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 + 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, 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	0	1	0	1	(1)	(1)	(1)	(1)	(1)	1	1	0	0	0	0	Rn					Rt				
size													Rs																		

32-bit (size == 10)

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

64-bit (size == 11)

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 <<~~integer s =~~ UInt(size);
integer regsize = if elsize == 64 then 64 else 32;~~(Rs);    // ignored by all loads and store-releaseAccT~~
~~integer elsize = 8 << UInt(size);~~
~~integer regsize = if elsize == 64 then 64 else 32;~~
~~integer datasize = elsize;~~
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) **htmldiff from-ISA_v84A_A64_xml_00bet7** **(new) ISA_v84A_A64_xml_00bet7_OPT**

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	1	0	0	0	1	0	1	(1)	(1)	(1)	(1)	(1)	1	1	0	0	0	0	Rn					Rt				
size													Rs																		

Integer

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);(Rt);
integer s =UInt(Rs);    // ignored by all loads and store-release

AccType acctype = AccType_ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1,[address, dbytes, acctype]; 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)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9-5	4-0
0	1	1	1	1	0	0	0	1	0	1	(1)	(1)	(1)	(1)	(1)	1	1	0	0	0	0	Rn	Rt
size											Rs												

Integer

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);(Rt);
integer s =UInt(Rs);    // ignored by all loads and store-release

AccType acctype = AccType_ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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);(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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	1	1	0	0	1	0	1	0	imm9									0	0	Rn					Rt				
size								opc																							

32-bit (size == 10)

```
LDAPUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
LDAPUR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
~~boolean wback = FALSE;~~
~~boolean postindex = FALSE;~~
integer scale = UInt(size);
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 regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;(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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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 AccType_ORDEREDMemOp_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;
    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.

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, 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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	0	0	1	1	0	0	1	0	1	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
LDAPURB <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);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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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 AccType_ORDEREDMemOp_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;
    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.

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, 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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	1	0	1	1	0	0	1	0	1	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
LDAPURH <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);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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 2,[address, datasize DIV 8, acctype] = data;

    when AccType_ORDEREDMemOp_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;
    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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	0	0	1	1	0	0	1	1	x	0	imm9	0	0	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDAPURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDAPURSB <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.

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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =     if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop !=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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

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_ORDERED] = data;

    when MemOp_LOAD
        data = Mem[address, 1,[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_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(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	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	x	0	imm9									0	0	Rn					Rt				
size								opc																							

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.

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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop !=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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

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_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(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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, 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*.

31	30	29	28	27	26	25	24	23	22	21	20-12	11	10	9-5	4-0
1	0	0	1	1	0	0	1	1	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
LDAPURSW <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);~~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 UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(32) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 4,[address, datasize DIV 8, acctype] = data;

    when AccType_ORDEREDMemOp_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;
    else
        X(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	9-5	4-0
1	x	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn	Rt
size									L		Rs					o0	Rt2						

32-bit (size == 10)

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

64-bit (size == 11)

```
LDAR <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;(Rt2); // ignored by load/store single register
integer s =UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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_ORDEREDMemOp_LOAD];[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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 n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data =case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1,[address, 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.

31	30	29	28	27	26	25	24	23	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	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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, 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.

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*.

31	30	29	28	27	26	25	24	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	Rt2					Rn					Rt				
									L		Rs					o0															

32-bit (sz == 0)

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit (sz == 1)

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 <<(Rt2); // ignored by load/store single register
integer s = UInt(sz);
integer datasize = elsize * 2;(Rs);    // ignored by all loads and store-releaseAccType acctype = if o0
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDAXP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if t == t2 thenif 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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 thenif 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       UnallocatedEncoding();
            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       UnallocatedEncoding();
            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;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
```

```
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

if 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_ORDERED];
    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(address, dbytes) then
                        iswrite = FALSE;
                        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault((acctype, iswrite, secondstage));AccType_ORDEREDX
    Mem[address + 0, 8, acctype];
                    X[t] =[t2] = Mem[address, 8,[address + 8, 8, acctype];
            else
                data = AccType_ORDEREDMem];[address, dbytes, acctype];
    X[t2] =[t] = MemZeroExtend[address+8, 8, AccType_ORDERED];(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.

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*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)			Rn					Rt		
size									L				Rs			o0			Rt2												

32-bit (size == 10)

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

64-bit (size == 11)

```
LDAXR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer 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 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
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 n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
            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      UnallocatedEncoding();
            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;
        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 = MemAccType_ORDEREDBigEndian];[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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 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.

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 n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
            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      UnallocatedEncoding();
            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;
        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);
(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>;
            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_ORDEREDBigEndian];[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	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	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);~~(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 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.

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
            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      UnallocatedEncoding();
            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;
        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, 2);
(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, 2,[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_ORDEREDBigEndian];[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.

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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
1	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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STCLR, STCLRL	A == '0' && Rt == '11111'

Operation

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

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 AND NOT(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	0	0	1	0	0	Rn	Rt
size													opc						

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>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STCLRB, STCLRLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

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 AND NOT(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, 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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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)

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

LDCLRAH (A == 1 && R == 0)

```
LDCLRAH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRALH (A == 1 && R == 1)

```
LDCLRALH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRH (A == 0 && R == 0)

```
LDCLRH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRLH (A == 0 && R == 1)

```
LDCLRLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STCLRH, STCLRLH	`A == '0' && Rt == '11111'`

Operation

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

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 AND NOT(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, 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.

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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	20-16	15	14	13	12	11	10	9-5	4-0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	0	1	0	0	0	Rn	Rt
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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STEOR, STEORL	`A == '0' && Rt == '11111'`

Operation

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

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

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	20-16	15	14	13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	0	1	0	0	0	Rn	Rt
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>]
```

LDEORLB (A == 0 && R == 1)

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STEORB, STEORLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

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
    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);(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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:~~45~~58

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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	20-16	15	14	13	12	11	10	9-5	4-0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	0	1	0	0	0	Rn	Rt
size													opc						

LDEORAH (A == 1 && R == 0)

```
LDEORAH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORALH (A == 1 && R == 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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STEORH, STEORLH	`A == '0' && Rt == '11111'`

Operation

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

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_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.

LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

32-bit (size == 10)

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

64-bit (size == 11)

```
LDLAR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer 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 << UInt(size);~~
~~integer regsize = if elsize == 64 then 64 else 32;~~
~~integer datasize = elsize;~~
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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_LIMITEDORDEREDMemOp_LOAD];[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) **htmldiff from-ISA_v84A_A64_xml_00bet7** **(new) ISA_v84A_A64_xml_00bet7_OPT**

LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset
(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)			Rn					Rt		
size									L				Rs			o0			Rt2												

No offset

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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 n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data =case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1,[address, dbytes, acctype] = data;

    when
        data = MemAccType_LIMITEDORDEREDMemOp_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	9-5	4-0
0	1	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn	Rt
size									L		Rs					o0	Rt2						

No offset

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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, dbytes, acctype] = data;

    when
        data = MemAccType_LIMITEDORDEREDMemOp_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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

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

31 30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1	0	1	1	0	0	0	1	imm7	Rt2	Rn	Rt
								L				

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.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then~~(Rt2); AccType acctype = AccType_VECSTREAM;~~
~~MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;~~
~~if opc == '11' then ~~UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

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   UnallocatedEncoding();
        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,~~[address + 0      , dbytes, acctype] = data1;~~ AccType_VECSTREAM~~Mem~~];
data2 =~~[address + dbytes, dbytes, acctype] = data2;~~

~~    when MemOp_LOAD~~
~~        data1 = ~~Mem[address+dbytes, dbytes,~~[address + 0      , dbytes, acctype];~~
~~        data2 = ~~AccType_VECSTREAM~~Mem~~];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;~~[address + dbytes, dbytes, acctype];~~
~~        if rt_unknown then~~
~~            data1 = bits(datasize) UNKNOWN;~~
~~            data2 = bits(datasize) UNKNOWN;~~
V[t] = data1;
V~~[t2] = data2;~~

~~if wback then~~
~~    if postindex then~~
~~        address = address + offset;~~
~~    if n == 31 then~~
~~        SP[] = address;~~
~~    else~~
        ~~X~~[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.

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*.

31	30	29	28	27	26	25	24	23	22	21–15	14–10	9–5	4–0
x	0	1	0	1	0	0	0	0	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
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

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then(Rt2); AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 thenif 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   UnallocatedEncoding();
        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_STREAMMem];
data2 =[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address+dbytes, dbytes,[address + 0      , dbytes, acctype];
        data2 = AccType_STREAMMem];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
X[t]  = data1;
        X[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[t] = data1;[] = 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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

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

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31 30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1	0	1	1	0	0	1	1	imm7	Rt2	Rn	Rt
								L				

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 = TRUE;
boolean postindex = TRUE;
```

Pre-index

31 30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1	0	1	1	0	1	1	1	imm7	Rt2	Rn	Rt
								L				

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 = TRUE;
boolean postindex = FALSE;
```

Signed offset

31 30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1	0	1	1	0	1	0	1	imm7	Rt2	Rn	Rt
								L				

32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP (SIMD&FP)*.

Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16. For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then(Rt2); AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 thenif 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   UnallocatedEncoding();
        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
        data1 = V[t];
        data2 = V[t2];
        Mem[address, dbytes,[address + 0      , dbytes, acctype] = data1; AccType_VECMem];
data2 =[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address+dbytes, dbytes,[address + 0      , dbytes, acctype];
        data2 = AccType_VECMem];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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

31	30	29	28	27	26	25	24	23	22	21–15	14–10	9–5	4–0
x	0	1	0	1	0	0	0	1	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21–15	14–10	9–5	4–0
x	0	1	0	1	0	0	1	1	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29	28	27	26	25	24	23	22	21–15	14–10	9–5	4–0
x	0	1	0	1	0	0	1	0	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then(Rt2); AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 thenif memop ==
    MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;     // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;     // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if t == t2 thenif memop ==
    MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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, dbytes,[address + 0      , dbytes, acctype] = data1; AccType_NORMALMem];
data2 =[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address+dbytes, dbytes,[address + 0      , dbytes, acctype];
        data2 = AccType_NORMALMem];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	21-15	14-10	9-5	4-0
0	1	1	0	1	0	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	21-15	14-10	9-5	4-0
0	1	1	0	1	0	0	1	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	21-15	14-10	9-5	4-0
0	1	1	0	1	0	0	1	0	1	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(Rt2);
bits(64) offset =(Rt2); AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
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);
```

Operation

```
bits(64) address;
bits(32) data1;
bits(32) data2;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 thenif memop ==
    MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;     // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if t == t2 thenif memop ==
    MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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(data2, 64);
[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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

31 30	29	28	27	26	25	24	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	1	0	imm9	0	1	Rn	Rt
							opc							

8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>], #<simm>
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>], #<simm>
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>], #<simm>
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>], #<simm>
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
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	1	0	imm9	1	1	Rn	Rt
							opc							

8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>, #<simm>]!
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, #<simm>]!
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>, #<simm>]!
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
size	1	1	1	1	0	1	x	1	imm12	Rn	Rt
							opc				

8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
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

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
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;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	20-12	11	10	9-5	4-0
1	x	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size								opc							

32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>], #<simm>
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20-12	11	10	9-5	4-0
1	x	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size								opc							

32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21-10	9-5	4-0
1	x	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size								opc				

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;(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
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
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
    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      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 thenif 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      UnallocatedEncoding();
        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;

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_NORMALMemOp_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(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.

31 30	29	28	27	26	25	24	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	1	1	Rm	option	S	1	0	Rn	Rt
							opc									

8-fsreg,LDR-8-fsreg (size == 00 && opc == 01 && option != 011)

```
LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

8-fsreg,LDR-8-fsreg (size == 00 && opc == 01 && option == 011)

```
LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-fsreg,LDR-16-fsreg (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-fsreg,LDR-32-fsreg (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-fsreg,LDR-64-fsreg (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-fsreg,LDR-128-fsreg (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding();     // 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 "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> For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

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”:

S	<amount>
0	#0
1	#2

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”:

S	<amount>
0	#0
1	#3

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”:

S	<amount>
0	#0
1	#4

Shared Decode

```
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

```
bits(64) offset = ExtendReg(m, extend_type, shift);
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;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8,[address, datasize DIV 8, acctype]; AccType_VECV];[t] = data

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        [] = address;
    else
        XVSP[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.

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	20–16	15–13	12	11	10	9–5	4–0
1	x	1	1	1	0	0	0	0	1	1	Rm	option	S	1	0	Rn	Rt
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>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding();     // 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”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<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”:

S	<amount>
0	#0
1	#2

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”:

S	<amount>
0	#0
1	#3

Shared Decode

```
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;(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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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 AccType_NORMALMemOp_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;
    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	20–12	11	10	9–5	4–0
1 1	1 1 1	0	0 0	M	S	1	imm9	W	1	Rn	Rt
size											

Key A, offset (M == 0 && W == 0)

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]
```

Key A, pre-indexed (M == 0 && W == 1)

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!
```

Key B, offset (M == 1 && W == 0)

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]
```

Key B, pre-indexed (M == 1 && W == 1)

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!
```

```
if !HavePACExt() then~~() || size != '11' then~~ UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
~~integer scale = 3;~~
bits(64) offset = LSL(SignExtend(S10, 64), 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|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.

Operation

```
bits(64) address;
bits(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      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if use_key_a then
    address = AuthDA(address, X[31]);
else
    address = AuthDB(address, X[31]);

address = address + offset;
data = Mem[address, 8, AccType_NORMAL];
X[t] = data;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20-12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size								opc							

Post-index

```
LDRB <Wt>, [<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	20-12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size								opc							

Pre-index

```
LDRB <Wt>, [<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	21-10	9-5	4-0
0	0	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size								opc				

Unsigned offset

```
LDRB <Wt>, [<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 *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

```
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
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
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 =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      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 thenif 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      UnallocatedEncoding();
        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;

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_NORMALMemOp_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(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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	0	1	1	Rm	option	S	1	0	Rn	Rt
size								opc									

Extended register (option != 011)

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then~~boolean wback = FALSE;~~
~~boolean postindex = FALSE;~~
~~integer scale = UInt(size);~~
~~if option<1> == '0' then~~ UnallocatedEncoding();     // sub-word 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.

<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”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
(m, extend_type, shift);
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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 AccType_NORMALMemOp_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;
    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.

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	1	0	0	0	0	1	0	imm9									0	1	Rn					Rt				
size								opc																							

Post-index

```
LDRH <Wt>, [<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	1	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn					Rt				
size								opc																							

Pre-index

```
LDRH <Wt>, [<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	1	1	1	1	0	0	1	0	1	imm12												Rn					Rt				
size								opc																							

Unsigned offset

```
LDRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
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.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
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
    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      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 thenif 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      UnallocatedEncoding();
        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;

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_NORMALMemOp_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(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.

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*.

31	30	29	28	27	26	25	24	23	22	21	20–16	15–13	12	11	10	9–5	4–0
0	1	1	1	1	0	0	0	0	1	1	Rm	option	S	1	0	Rn	Rt
size								opc									

32-bit

```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

`if option<1> == '0' then`~~`boolean wback = FALSE;`~~
~~`boolean postindex = FALSE;`~~
~~`integer scale = UInt(size);`~~
~~`if option<1> == '0' then `~~`UnallocatedEncoding();     // 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;`~~

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<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”:

S	<amount>
0	#0
1	#1

Shared Decode

```
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 UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 2,[address, datasize DIV 8, acctype] = data;

    when AccType_NORMALMemOp_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;
    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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	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	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												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;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =     if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(8) data;
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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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(8) UNKNOWN;
            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, 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 wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	1	x	1	Rm	option	S	1	0	Rn	Rt
size								opc									

32-bit with extended register offset (opc == 11 && option != 011)

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

32-bit with shifted register offset (opc == 11 && option == 011)

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

64-bit with extended register offset (opc == 10 && option != 011)

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

64-bit with shifted register offset (opc == 10 && option == 011)

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' thenboolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding();     // sub-word 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”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
(m, extend_type, shift);
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop !=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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

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_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(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	0	imm9									0	1	Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>], #<simm>
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
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	1	0	0	0	1	x	0	imm9									1	1	Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
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	1	0	0	1	1	x	imm12												Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
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 *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;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =    if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(16) data;
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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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(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, 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 wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-13	12	11	10	9-5	4-0
0	1	1	1	1	0	0	0	1	x	1	Rm	option	S	1	0	Rn	Rt
size								opc									

32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then~~boolean wback = FALSE;~~
~~boolean postindex = FALSE;~~
~~integer scale = UInt(size);~~
~~if option<1> == '0' then ~~UnallocatedEncoding();     // 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;~~
```

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”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<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”:

S	<amount>
0	#0
1	#1

Shared Decode

```
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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop !=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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

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_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(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	imm9									0	1	Rn					Rt				
size								opc																							

Post-index

```
LDRSW <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
1	0	1	1	1	0	0	0	1	0	0	imm9									1	1	Rn					Rt				
size								opc																							

Pre-index

```
LDRSW <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
1	0	1	1	1	0	0	1	1	0	imm12												Rn					Rt				
size								opc																							

Unsigned offset

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset =~~integer scale = ~~ ~~UInt(size);~~
~~bits(64) offset = ~~LSL(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.
<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.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
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 =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      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 thenif 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      UnallocatedEncoding();
        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;

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_NORMALMemOp_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(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.

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 =(Rt); MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<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,[address, size, AccType_NORMAL];];
        if signed then
X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCHPrefetch(data, 64);(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-13	12	11	10	9-5	4-0
1	0	1	1	1	0	0	0	1	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

64-bit

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

if option<1> == '0' then~~boolean wback = FALSE;~~
~~boolean postindex = FALSE;~~
~~integer scale = UInt(size);~~
~~if option<1> == '0' then~~ UnallocatedEncoding(); // 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|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”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<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”:

S	<amount>
0	#0
1	#2

Shared Decode

```
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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(32) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 4,[address, datasize DIV 8, acctype] = data;

    when AccType_NORMALMemOp_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;
    else
        X(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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	0	A	R	1	Rs					0	0	1	1	0	0	Rn					Rt				
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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STSET, STSETL	`A == '0' && Rt == '11111'`

Operation

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

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSETB, STSETLB.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	0	1	1	0	0	Rn	Rt
size													opc						

LDSETAB (A == 1 && R == 0)

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALB (A == 1 && R == 1)

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETB (A == 0 && R == 0)

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLB (A == 0 && R == 1)

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STSETB, STSETLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

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_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);(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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.
- `LDSETLH` 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.

Integer

(ARMv8.1)

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

LDSETAH (A == 1 && R == 0)

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALH (A == 1 && R == 1)

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETH (A == 0 && R == 0)

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLH (A == 0 && R == 1)

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STSETH, STSETLH	`A == '0' && Rt == '11111'`

Operation

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

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 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, 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
- `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
- `LDSMAX` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMAX, STSMAXL.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	1	0	0	0	0	Rn	Rt
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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STSMAX, STSMAXL	`A == '0' && Rt == '11111'`

Operation

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

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 = ifcase 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;(value) then value
Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	1	0	0	0	0	Rn	Rt
size													opc						

LDSMAXAB (A == 1 && R == 0)

```
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXALB (A == 1 && R == 1)

```
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXB (A == 0 && R == 0)

```
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXLB (A == 0 && R == 1)

```
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STSMAXB, STSMAXLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

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 = ifcase 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;(value) then 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:~~45~~58

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	20-16	15	14	13	12	11	10	9-5	4-0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	1	0	0	0	0	Rn	Rt
size													opc						

LDSMAXAH (A == 1 && R == 0)

```
LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXALH (A == 1 && R == 1)

```
LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXH (A == 0 && R == 0)

```
LDSMAXH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXLH (A == 0 && R == 1)

```
LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STSMAXH, STSMAXLH	`A == '0' && Rt == '11111'`

Operation

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

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 = ifcase 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;(value) then value
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	0	Rn	Rt
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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STSMIN, STSMINL	`A == '0' && Rt == '11111'`

Operation

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

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 = ifcase op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;
Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMINB, STSMINLB.

Integer
(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	1	0	1	0	0	Rn	Rt
size													opc						

LDSMINAB (A == 1 && R == 0)

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALB (A == 1 && R == 1)

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINB (A == 0 && R == 0)

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLB (A == 0 && R == 1)

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STSMINB, STSMINLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

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 = ifcase 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);(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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMINH, STSMINLH.

Integer

(ARMv8.1)

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

LDSMINAH (A == 1 && R == 0)

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALH (A == 1 && R == 1)

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINH (A == 0 && R == 0)

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLH (A == 0 && R == 1)

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STSMINH, STSMINLH	`A == '0' && Rt == '11111'`

Operation

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

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 = ifcase 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, 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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
1	x	1	1	1	0	0	0	0	1	0	imm9	1	0	Rn	Rt
size								opc							

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.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;
integer regsize;

regsize = if size == '11' then 64 else 32;
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
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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];[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
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    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.

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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	0	1	1	1	0	0	0	0	1	0	imm9	1	0	Rn	Rt
size								opc							

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    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
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 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_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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	1	1	1	1	0	0	0	0	1	0	imm9	1	0	Rn	Rt
size								opc							

Unscaled offset

```
LDTRH <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);
AccType acctype = AccType_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    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~~
~~        UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 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_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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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
0	0	1	1	1	0	0	0	1	x	0	imm9	1	0	Rn	Rt
size								opc							

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.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    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;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =     if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop !=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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

case memop of
    when MemOp_STORE
        data =if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, acctype] = data;
[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, 1, acctype];
[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;
    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.

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
0	1	1	1	1	0	0	0	1	x	0	imm9									1	0	Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDTRSH <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.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    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;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =    if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop !=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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

case memop of
    when MemOp_STORE
        data =if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        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);

    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(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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20-12	11	10	9-5	4-0
1	0	1	1	1	0	0	0	1	0	0	imm9	1	0	Rn	Rt
size								opc							

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);
AccType acctype = AccType_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    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~~
~~        UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(32) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 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);

    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, 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.

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	20–16	15	14	13	12	11	10	9–5	4–0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	1	1	0	0	0	Rn	Rt
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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STUMAX, STUMAXL	`A == '0' && Rt == '11111'`

Operation

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

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 = ifcase 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;(value) then value
Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	20-16	15	14	13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	1	1	0	0	0	Rn	Rt
size													opc						

LDUMAXAB (A == 1 && R == 0)

```
LDUMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXALB (A == 1 && R == 1)

```
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXB (A == 0 && R == 0)

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXLB (A == 0 && R == 1)

```
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STUMAXB, STUMAXLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

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 = ifcase 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;(value) then 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	20-16	15	14	13	12	11	10	9-5	4-0
0	1	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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STUMAXH, STUMAXLH	`A == '0' && Rt == '11111'`

Operation

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

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 = ifcase 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;(value) then value
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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
1	x	1	1	1	0	0	0	A	R	1	Rs					0	1	1	1	0	0	Rn					Rt				
size																opc															

32-bit LDUMIN (size == 10 && A == 0 && R == 0)

```
LDUMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINA (size == 10 && A == 1 && R == 0)

```
LDUMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINAL (size == 10 && A == 1 && R == 1)

```
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINL (size == 10 && A == 0 && R == 1)

```
LDUMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDUMIN (size == 11 && A == 0 && R == 0)

```
LDUMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINA (size == 11 && A == 1 && R == 0)

```
LDUMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINAL (size == 11 && A == 1 && R == 1)

```
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINL (size == 11 && A == 0 && R == 1)

```
LDUMINL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STUMIN, STUMINL	`A == '0' && Rt == '11111'`

Operation

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

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 = ifcase 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.

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	20-16	15	14	13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	1	1	1	0	0	Rn	Rt
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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STUMINB, STUMINLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

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 = ifcase 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);(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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	1	0	0	0	A	R	1	Rs	0	1 1 1	0	0	Rn	Rt
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 UnallocatedEncoding();
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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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

Alias	Is preferred when
STUMINH, STUMINLH	`A == '0' && Rt == '11111'`

Operation

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

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 = ifcase 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, 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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:~~45~~58

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31 30	29	28	27	26	25	24	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	1	0	imm9	0	0	Rn	Rt
							opc							

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 UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
~~AccType acctype = AccType_VEC;~~
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
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;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8,[address, datasize DIV 8, acctype]; AccType_VECV];[t] = data

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        [] = address;
    else
        XVSP[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20-12	11	10	9-5	4-0
1	x	1	1	1	0	0	0	0	1	0	imm9	0	0	Rn	Rt
size								opc							

32-bit (size == 10)

```
LDUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
LDUR <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 regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;(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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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 AccType_NORMALMemOp_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;
    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.

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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	0	1	1	1	0	0	0	0	1	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
LDURB <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);~~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 UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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 AccType_NORMALMemOp_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;
    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.

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*.

31	30	29	28	27	26	25	24	23	22	21	20-12	11	10	9-5	4-0
0	1	1	1	1	0	0	0	0	1	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
LDURH <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);~~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 UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 2,[address, datasize DIV 8, acctype] = data;

    when AccType_NORMALMemOp_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;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	0	1	1	1	0	0	0	1	x	0	imm9	0	0	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDURSB <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.

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;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop !=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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

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_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(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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	22	21	20-12	11	10	9-5	4-0
0	1	1	1	1	0	0	0	1	x	0	imm9	0	0	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDURSH <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.

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;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop =    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    if memop !=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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

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_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(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	0	1	1	1	0	0	0	1	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
LDURSW <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);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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(32) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 4,[address, datasize DIV 8, acctype] = data;

    when AccType_NORMALMemOp_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;
    else
        X(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.

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)	0	Rt2					Rn					Rt				
									L		Rs					o0															

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 <<(Rt2); // ignored by load/store single register
integer s = UInt(sz);
integer datasize = elsize * 2;(Rs);   // ignored by all loads and store-releaseAccType acctype = if o0
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDXP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if t == t2 thenif 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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 thenif 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       UnallocatedEncoding();
            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       UnallocatedEncoding();
            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;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
```

```
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

if 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>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault((acctype, iswrite, secondstage));AccType_ATOMICX,
    Mem[address + 0, 8, acctype];
                X[t] =[t2] = Mem[address, 8,[address + 8, 8, acctype];
            else
                data = AccType_ATOMICMem];[address, dbytes, acctype];
    X[t2] =[t] = MemZeroExtend[address+8, 8, AccType_ATOMIC];(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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9–5	4–0
1	x	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn	Rt
size									L		Rs					o0	Rt2						

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 <<~~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 == '1' then AccType_ORDERED else AccType_ATOMIC;~~
~~boolean pair = FALSE;~~
~~MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;~~
~~integer elsize = 8 << UInt(size);~~
~~integer regsize = if elsize == 64 then 64 else 32;~~
~~integer datasize = if pair then elsize * 2 else elsize;~~
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
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 n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
            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      UnallocatedEncoding();
            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 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.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 = MemAccType_ATOMICBigEndian];[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);~~(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 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.

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 n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
            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      UnallocatedEncoding();
            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;
        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);
(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>;
            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_ATOMICBigEndian];[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9-5	4-0
0	1	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn	Rt
size									L		Rs					o0	Rt2						

No offset

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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 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.

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
            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      UnallocatedEncoding();
            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;
        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, 2);
(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, 2,[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_ATOMICBigEndian];[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	1	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>
```

```
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 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

Alias	Is preferred when
MUL	`Ra == '11111'`

Operation

```
bits(destsize) operand1 =bits(datasize) operand1 = X[n];
bits(destsize) operand2 =bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
    result = UInt(operand3) + ((operand3) - (UInt(operand1) * UInt(operand2));(operand2));
else
    result =

UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d] = result<destsize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:~~45~~58

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23 22	21	20	19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	0	0	0	0	H	0	Rn	Rd
													o2						

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 UnallocatedEncoding();

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”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23-22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	0	1	0	1	Rn	Rd
		U																

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 ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23-22	21	20	19-16	15	14	13	12	11	10	9-5	4-0
0	Q	1	0	1	1	1	1	size	L	M	Rm	0	1	0	0	H	0	Rn	Rd
													o2						

Vector

```
MLS <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 UnallocatedEncoding();

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”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	1	0	size	1	Rm	1	0	0	1	0	1	Rn	Rd
		U																

Three registers of the same type

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

31	30	29	28	27	26	25	24	23	22 21	20 … 5	4 … 0
sf	1	1	1	0	0	1	0	1	hw	imm16	Rd
	opc										

32-bit (sf == 0)

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVK <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' thenbits(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 UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
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

```
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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).

31	30	29	28	27	26	25	24	23	22 21	20 … 5	4 … 0
sf	0	0	1	0	0	1	0	1	hw	imm16	Rd
	opc										

32-bit (sf == 0)

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

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

if sf == '0' && hw<1> == '1' then~~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 UnallocatedEncoding();~~

~~if sf == '0' && hw<1> == '1' then ~~UnallocatedEncoding();
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

Alias	Of variant	Is preferred when
MOV (inverted wide immediate)	64-bit	`! (IsZero(imm16) && hw != '00')`
MOV (inverted wide immediate)	32-bit	`! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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).

31	30	29	28	27	26	25	24	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	hw		imm16																Rd				
	opc																														

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~~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 UnallocatedEncoding();~~

~~if sf == '0' && hw<1> == '1' then~~ UnallocatedEncoding();
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

Alias	Is preferred when
MOV (wide immediate)	`! (IsZero(imm16) && hw != '00')`

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;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	18-16	15-12	11-8	7-5	4-0
1	1	0	1	0	1	0	1	0	0	1	1	o0	op1	CRn	CRm	op2	Rt
										L							

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);~~(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”:

o0	<op0>
0	2
1	3

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

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

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

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

Operation

~~if read then~~
~~X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);~~
~~else~~
~~AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2,~~ X[t] = AArch64.SysRegRead(sys_op0,

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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 are D, A, I, F, and SP. This set of bits is expanded in extensions to the architecture as follows:

- *ARMv8.1-PAN* adds the PAN bit.
- *ARMv8.2-UAO* adds the UAO bit.
- *ARMv8.4-DIT* adds the DIT bit.

31	30	29	28	27	26	25	24	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	op1			0	1	0	0	CRm				op2			1	1	1	1	1

System

```
MSR <pstatefield>, #<imm>
```

```
AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');~~('00', op1, '0100', CRm, op2, '111~~

~~bits(4) operand = CRm;~~

PSTATEField field;
case op1:op2 of
    when '000 000'
        SEE "CFINV";
    when '000 011'
        if !HaveUAOExt() then
            UnallocatedEncoding();
        field = PSTATEField_UAO;
    when '000 100'
        if !HavePANExt() then
            UnallocatedEncoding();
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '011 010'
        if !HaveDITExt() then
            UnallocatedEncoding();
        field = PSTATEField_DIT;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    otherwise UnallocatedEncoding();

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if op1 == '011' && PSTATE.EL == EL0 && (IsInHost() || SCTLR_EL1.UMA == '0') then
    AArch64.SystemRegisterTrap(EL1, '00', op2, op1, '0100', '11111', CRm, '0');
```

Assembler Symbols

<pstatefield> Is a PSTATE field name, encoded in “op1:op2”:

op1	op2	<pstatefield>	Architectural Feature
000	011	UAO	ARMv8.2-UAO
000	100	PAN	ARMv8.1-PAN
000	101	SPSel	-
011	010	DIT	ARMv8.4-DIT
011	110	DAIFSet	-
011	111	DAIFClr	-

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
case field of
    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>;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3 2 1 0
1	1	0	1	0	1	0	1	0	0	0	1	o0	op1	CRn	CRm	op2	Rt
										L							

System

```
MSR (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>
```

```
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);(CRm);
boolean read = (L == '1');
```

Assembler Symbols

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".
The System register names are defined in *'AArch64 System Registers' in the System Register XML*.

<op0> Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

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

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

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

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

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

Operation

```
if read then
    X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14-10	9-5	4-0
sf	0	0	1	1	0	1	1	0	0	0	Rm	1	Ra	Rn	Rd
												o0			

32-bit (sf == 0)

```
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit (sf == 1)

```
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

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

Alias	Is preferred when
MNEG	`Ra == '11111'`

Operation

```
bits(destsize) operand1 =bits(datasize) operand1 = X[n];
bits(destsize) operand2 =bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
    result = UInt(operand3) - (UInt(operand1) * UInt(operand2));(operand2));
else
    result =
UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d] = result<destsize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	19-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	1	size	L	M	Rm	1	0	0	0	H	0	Rn	Rd

Vector

```
MUL <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 UnallocatedEncoding();

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;
```

Assembler Symbols

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

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

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
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>;    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

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

Three registers of the same type

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then ReservedValue();
if size == '11' then ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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)*(element1) *UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | **htmldiff from-ISA_v84A_A64_xml_00bet7** | **(new) ISA_v84A_A64_xml_00bet7_OPT**

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	15-12	11	10	9	8	7	6	5	4-0
0	Q	1	0	1	1	1	1	0	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 = when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx11' operation = when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x01' operation = when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x11' operation = when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x1' operation = when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '1110x' operation = when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        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 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.
For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.
For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

Operation

```
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.

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

System

```
NOP
```

```
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 111'
        SEE "XPACLRI";
    when '0001 xxx'
        SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
    when '0010 000'
        if !HaveRASExt() then EndOfInstruction();                  // Instruction executes as NOP
        op = SystemHintOp_ESB;
    when '0010 001'
        if !HaveStatisticalProfiling() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_PSB;
    when '0010 010'
        if !HaveSelfHostedTrace() then EndOfInstruction();         // Instruction executes as NOP
        op = SystemHintOp_TSB;
    when '0010 100'
        op = SystemHintOp_CSDB;
    when '0011 xxx'
        SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    otherwise  EndOfInstruction// Empty.();                          // Instruction executes as
```

Operation

```
// do nothingcase op of
    whenSystemHintOp_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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

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

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	0	1	1	1	Rm	0	0	0	1	1	1	Rn	Rd
								size											

Three registers of the same type

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

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

boolean invert = (size<0> == '1');LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AN
```

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

```
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 OR operand2;case op of
    when

LogicalOp_AND
        result = operand1 AND operand2;
    when LogicalOp_ORR
        result = operand1 OR operand2;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

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

ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	1	0	1	0	1	0	shift	1	Rm	imm6	Rn	Rd
	opc								N				

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' thenboolean 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 ReservedValue();

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<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

Alias	Is preferred when
MVN	Rn == '11111'

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ORR (vector, immediate)

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

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
		op															cmode														

16-bit (cmode == 10x1)

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

32-bit (cmode == 0xx1)

```
ORR <Vd>.<T>, #<imm8>{, LSL #<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 '0xx00' operation = ImmediateOp_MOVI;
    when '0xx10' operation =    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '10x00' operation =    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x10' operation =    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '110x0' operation =    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '1110x' operation =    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 UnallocatedEncoding;
    when '11110' operation =();
        operation = ImmediateOp_MOVI;
imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

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

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.
For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

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

This instruction is used by the alias MOV (vector).

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	0	1	0	1	Rm	0	0	0	1	1	1	Rn	Rd
								size											

Three registers of the same type

```
ORR <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 datasize = if Q == '1' then 128 else 64;~~
~~integer elements = datasize DIV esize;~~

~~boolean invert = (size<0> == '1');LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AN~~
```

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.

Alias Conditions

Alias	Is preferred when
MOV (vector)	`Rm == Rn`

Operation

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

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;

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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).

31	30	29	28	27	26	25	24	23	22	21-16	15-10	9-5	4-0
sf	0	1	1	0	0	1	0	0	N	immr	imms	Rn	Rd
	opc												

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

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

64-bit (sf == 1)

```
ORR <Xd|SP>, <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' thenboolean 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 ReservedValue();
(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

Alias	Is preferred when
MOV (bitmask immediate)	`Rn == '11111' && ! MoveWidePreferred(sf, N, imms, immr)`

Operation

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

result = operand1 OR imm;
if d == 31 thencase 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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).

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	1	0	1	0	1	0	shift	0	Rm	imm6	Rn	Rd
	opc								N				

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<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

Alias	Is preferred when
MOV (register)	`shift == '00' && imm6 == '000000' && Rn == '11111'`

Operation

```
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 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

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

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 ReservedValue();
if size == '11' then ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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)*(element1) *UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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
1	1	1	1	1	0	0	1	1	0	imm12	Rn	Rt
size								opc				

Unsigned offset

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

`bits(64) offset =`~~`boolean wback = FALSE;`~~
~~`boolean postindex = FALSE;`~~
~~`integer scale = UInt(size);`~~
~~`bits(64) offset = `~~`LSL(ZeroExtend(imm12, 64), 3);`~~`(imm12, 64), scale);`~~

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.

<pimm> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    address =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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

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
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    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 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	23–5	4–0
1	1	0	1	1	0	0	0	imm19	Rt
opc									

Literal

```
PRFM (<prfop>|#<imm5>), <label>
```

```
integer t = UInt(Rt);
bits(64) offset;

offset =(Rt); MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<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>.
<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;[] + 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_PREFETCHPrefetch(address, t<4:0>);
```

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*.

31	30	29	28	27	26	25	24	23	22	21	20–16	15–13	12	11	10	9–5	4–0
1	1	1	1	1	0	0	0	1	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

Integer

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

`if option<1> == '0' then`~~`boolean wback = FALSE;`~~
~~`boolean postindex = FALSE;`~~
~~`integer scale = UInt(size);`~~
~~`if option<1> == '0' then `~~`UnallocatedEncoding();     // 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.

<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”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<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”:

S	<amount>
0	#0
1	#3

Shared Decode

```
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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    address =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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

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
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(address, t<4:0>);[n] = address;
```

PRFM (unscaled offset)

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
1	1	1	1	1	0	0	0	1	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]
```

`bits(64) offset =`~~`boolean wback = FALSE;`~~
~~`boolean postindex = FALSE;`~~
~~`integer scale = UInt(size);`~~
~~`bits(64) offset = `~~`SignExtend(imm9, 64);`

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD
Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI
Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST
Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1
Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2
Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3
Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP
Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM
Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.
For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.
This syntax is only for encodings that are not accessible using <prfop>.

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

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 then
    address =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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;

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
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(address, t<4:0>);[n] = address;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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)

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

System

```
PSB CSYNC
```

```
SystemHintOpif !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() then;
    when '0011 xxx'
        SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    otherwise EndOfInstruction();();                                 // Instruction executes as NOP
```

Operation

```
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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();();

    otherwise // do nothing
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

PSSBB

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.

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

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 UnallocatedEncoding();

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> == '00' then
            op = MemBarrierOp_SSBB;
        elsif CRm<3:2> == '01' then
            op = MemBarrierOp_PSSBB;
        else
            types = MBReqTypes_All;
            domain = MBReqDomain_FullSystem// Empty.;
```

Operation

```
case op of
    when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);
    when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
    when MemBarrierOp_ISBInstructionSynchronizationBarrier();
    when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
    when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
```

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	0	1	1	1	0	0	1	1	Rm					1	0	0	0	1	1	Rn					Rd				

Advanced SIMD

```
RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D
```

```
if !HaveSHA3Ext() then UnallocatedEncoding();
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):(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:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

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

Integer

```
RET {<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
    UnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n !=31 then UnallocatedEncoding();
        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

```
bits(64) target = X[n];[n];
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_CALL then X[30] = PC[] + 4;
BranchTo(target, BranchType_RET);(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	0	1	0	1	1	1	1	1	0	0	0	0	1	M	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
    UnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET() thenthen
        if n !=31 then
    UnallocatedEncoding();();
        n = 30;
        source_is_sp = TRUE;
```

Operation

```
bits(64) target = X[30];
bits(64) modifier =[n];
if pac then
    bits(64) modifier = if source_is_sp then SP[];

if use_key_a then
    target =[] else X[m];

    if use_key_a then
        target = AuthIA(target, modifier);
else
    target = AuthIB(target, modifier);(target, modifier);

if branch_type ==

BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, BranchType_RET);(target, branch_type);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64.

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

32-bit (sf == 0 && opc == 10)

```
REV <Wd>, <Wn>
```

64-bit (sf == 1 && opc == 11)

```
REV <Xd>, <Xn>
```

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

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        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>;
        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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	0	0	1	1	0	Rn					Rd				
		U																	o0												

Vector

```
REV16 <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)
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) + (op) + UInt(size) >= 3 then UnallocatedEncoding();

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>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	0	0	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 UnallocatedEncoding();
        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>;
        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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	0	0	0	0	1	0	Rn					Rd				
		U																	o0												

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)
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) +(op) + UInt(size) >= 3 then UnallocatedEncoding();

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>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
1x	x	RESERVED

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word 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
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn					Rd				
sf																				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 UnallocatedEncoding();
        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>;
        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.

31	30	29	28	27	26	25	24	23	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	0	0	0	1	0	Rn					Rd				
		U																	o0												

Vector

```
REV64 <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)
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) +(op)+ UInt(size) >= 3 then UnallocatedEncoding();

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>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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
1	0	1	1	1	0	1	0	0	0	0	imm6						0	0	0	0	1	Rn					0	mask			
sf																															

Integer

```
RMIF <Xn>, #<shift>, #<mask>
```

```
if !HaveFlagManipulateExt() then() || sf != '1' then UnallocatedEncoding();
integer lsb = UInt(imm6);
integer n = UInt(Rn);
```

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
<mask>	Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

Operation

```
bits(4) tmp;
bits(64) tmpreg = X[n];
tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
if mask<3> == '1' then PSTATE.N = tmp<3>;
if mask<2> == '1' then PSTATE.Z = tmp<2>;
if mask<1> == '1' then PSTATE.C = tmp<1>;
if mask<0> == '1' then PSTATE.V = tmp<0>;
```

Operational information

If PSTATE.DIT is 1:

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

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.

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

Three registers of the same type

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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>;(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

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

Three registers, not all the same type

```
SABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

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

if size == '11' then ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

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

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

Three registers of the same type

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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>;(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.

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

Three registers, not all the same type

```
SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

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

if size == '11' then ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	0	1	1	1	0	size	1	0	0	0	0	0	0	1	1	0	1	0	Rn	Rd
		U														op						

Vector

```
SADALP <Vd>.<Ta>, <Vn>.<Tb>
```

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

if size == '11' then ReservedValue();
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”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

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

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

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
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>;    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

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

Vector

```
SADDLP <Vd>.<Ta>, <Vn>.<Tb>
```

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

if size == '11' then ReservedValue();
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":

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

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

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

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
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>;    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
sf	1	0	1	1	0	1	0	0	0	0	Rm	0	0	0	0	0	0	Rn	Rd
	op	S																	

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

Alias	Is preferred when
NGC	`Rn == '11111'`

Operation

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

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

(result, -) =(result, nzcv) = 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
sf	1	1	1	1	0	1	0	0	0	0	Rm	0	0	0	0	0	0	Rn	Rd
	op	S																	

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 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

Alias	Is preferred when
NGCS	Rn == '11111'

Operation

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

operand2 = NOT(operand2);
~~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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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 (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW.

31	30	29	28	27	26	25	24	23	22	21-16	15-10	9-5	4-0
sf	0	0	1	0	0	1	1	0	N	immr	imms	Rn	Rd
	opc												

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

```
SBFM <Wd>, <Wn>, #<immr>, #<imms>
```

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

```
SBFM <Xd>, <Xn>, #<immr>, #<imms>
```

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

~~boolean inzero;~~
~~boolean extend;~~
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then~~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' UnallocatedEncoding();~~

~~if sf == '1' && N != '1' then ~~ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

Assembler Symbols

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

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

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

Alias Conditions

Alias	Of variant	Is preferred when
ASR (immediate)	32-bit	`imms == '011111'`
ASR (immediate)	64-bit	`imms == '111111'`
SBFIZ		`UInt(imms) < UInt(immr)`
SBFX		`BFXPreferred(sf, opc<1>, imms, immr)`
SXTB		`immr == '000000' && imms == '000111'`
SXTH		`immr == '000000' && imms == '001111'`
SXTW		`immr == '000000' && imms == '011111'`

Operation

bits(datasize) src =~~bits(datasize) dst = if inzero then~~ ~~Zeros() else X[d];~~
~~bits(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;
~~(src, R) AND wmask);~~

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

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

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

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

rounding =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
        UnallocatedEncoding(FPCR);();
```

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, FALSE, FPCR, rounding);(intval, fracbits, unsigned, FPCR, rounding
V[d] = fltval;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

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

32-bit to half-precision (sf == 0 && type == 11)
(ARMv8.2)

```
SCVTF <Hd>, <Wn>
```

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

```
SCVTF <Sd>, <Wn>
```

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

```
SCVTF <Dd>, <Wn>
```

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

```
SCVTF <Hd>, <Xn>
```

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

```
SCVTF <Sd>, <Xn>
```

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

```
SCVTF <Dd>, <Xn>
```

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'    when '10'
        if opcode<2:1>:rmode != '11 01' then
        UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding(FPCR);();
```

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

```
CheckFPAdvSIMDEnabled64();

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<fltsize-1:0>;
        Vpart[d,part] = fltval;
    when FPConvOp_CVT_FtoI_JS
        fltval = V[n];
        intval = FPToFixedJS(fltval, FPCR, TRUE);
        X[d] = ZeroExtend[d] = fltval;(intval<31:0>, 64);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	0	0	1	1	Rn	Rd
																	o1		

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(operan

X[d] = result<datasize-1:0>;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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
0	Q	0	0	1	1	1	1	size	L	M	Rm	1	1	1	0	H	0	Rn	Rd
		U																	

Vector

```
SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then ReservedValue();
boolean signed = (U == '0');
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

```
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]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
[operand2, 4 * index + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
[operand2, 4 * index + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```


ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

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

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Three registers of the same type

(ARMv8.2)

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

Three registers of the same type

```
SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' thenif size!= '10' then ReservedValue();
boolean signed = (U == '0');
boolean signed = (U=='0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

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

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

Q	<Ta>
0	2S
1	4S

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

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

Q	<Tb>
0	8B
1	16B

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

Operation

```
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]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
[operand2, 4 * e + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SETF8, SETF16

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. SETF8 treats the value as an 8 bit value, and SETF16 treats the value as an 16 bit value.

The PSTATE.C flag is not affected by these instructions.

Integer
(ARMv8.4)

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

SETF8 (sz == 0)

```
SETF8 <Wn>
```

SETF16 (sz == 1)

```
SETF16 <Wn>
```

```
if !HaveFlagManipulateExt() then~~() || sf != '0' then~~ UnallocatedEncoding();
integer msb = if sz == '1' then 15 else 7;
~~integer msb = if sz=='1' then 15 else 7;~~
integer n = UInt(Rn);
```

Assembler Symbols

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

Operation

```
bits(32) tmpreg = X[n];
PSTATE.N = tmpreg<msb>;
PSTATE.Z = if (tmpreg<msb:0> == Zeros(msb + 1)) then '1' else '0';
~~(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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*.

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

System

```
SEV
```

```
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 111'
        SEE "XPACLRI";
    when '0001 xxx'
        SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
    when '0010 000'
        if !HaveRASExt() then EndOfInstruction();                   // Instruction executes as NOP
        op = SystemHintOp_ESB;
    when '0010 001'
        if !HaveStatisticalProfiling() then EndOfInstruction();     // Instruction executes as NOP
        op = SystemHintOp_PSB;
    when '0010 010'
        if !HaveSelfHostedTrace() then EndOfInstruction();          // Instruction executes as NOP
        op = SystemHintOp_TSB;
    when '0010 100'
        op = SystemHintOp_CSDB;
    when '0011 xxx'
        SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    otherwise  EndOfInstruction// Empty.();                                  // Instruction executes as
```

Operation

```
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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();();

    otherwise // do nothing
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

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

System

```
SEVL
```

```
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";
    otherwise  EndOfInstruction// Empty.();                              // Instruction executes as
```

Operation

```
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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();();

    otherwise // do nothing
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

SHA1C

SHA1 hash update (choose).

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

Advanced SIMD

```
SHA1C <Qd>, <Sn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UnallocatedEncoding();
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32)  Y = V[n];     // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32)  t;

for e = 0 to 3
    t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SHA1M

SHA1 hash update (majority).

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	1	0	1	1	1	1	0	0	0	0	Rm	0	0	1	0	0	0	Rn	Rd

Advanced SIMD

```
SHA1M <Qd>, <Sn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UnallocatedEncoding();
```

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);(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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	0	Rm					0	0	0	1	0	0	Rn					Rd				

Advanced SIMD

```
SHA1P <Qd>, <Sn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UnallocatedEncoding();
```

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SHA1SU0

SHA1 schedule update 0.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	1	0	1	1	1	1	0	0	0	0	Rm	0	0	1	1	0	0	Rn	Rd

Advanced SIMD

```
SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UnallocatedEncoding();
```

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	1	0	0	0	0	Rm					0	1	0	1	0	0	Rn					Rd				
																			P												

Advanced SIMD

```
SHA256H2 <Qd>, <Qn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UnallocatedEncoding();();
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],[d], V[d],[n], V[m], FALSE);[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SHA256H

SHA256 hash update (part 1).

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

Advanced SIMD

```
SHA256H <Qd>, <Qn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UnallocatedEncoding();();
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

SHA256SU0

SHA256 schedule update 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
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	Rn					Rd				

Advanced SIMD

```
SHA256SU0 <Vd>.4S, <Vn>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UnallocatedEncoding();
```

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 = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SHA256SU1

SHA256 schedule update 1.

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

Advanced SIMD

```
SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UnallocatedEncoding();
```

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 = ROR(elt, 17) EOR ROR(elt, 19) EOR 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];
[T1, e - 2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR 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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

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

Advanced SIMD

(ARMv8.2)

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

Advanced SIMD

```
SHA512H2 <Qd>, <Qn>, <Vm>.2D
```

```
if !HaveSHA512Ext() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(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(Y<63:0>,34) EOR ROR(Y<63:0>, 39);
(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>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 +  W<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR(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>);(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register. This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
1	1	0	0	1	1	1	0	0	1	1	Rm	1	0	0	0	0	0	Rn	Rd

Advanced SIMD

```
SHA512H <Qd>, <Qn>, <Vm>.2D
```

```
if !HaveSHA512Ext() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) MSigma1;
bits(64) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

MSigma1 = ROR(Y<127:64>, 14) EOR ROR(Y<127:64>, 18) EOR(Y<127:64>,18) EOR ROR(Y<127:64>, 41);
Vtmp<127:64> = (Y<127:64> AND X<63:0>) EOR (NOT(Y<127:64>) AND X<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + W<127:64>);
(Y<127:64>,41);
Vtmp<127:64> = (Y<127:64> AND X<63:0>) EOR (NOT(Y<127:64>) AND X<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + W<127:64>);
tmp = Vtmp<127:64> + Y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR(tmp,18) EOR ROR(tmp, 41);
(tmp,41);
Vtmp<63:0> = (tmp AND Y<127:64>) EOR (NOT(tmp) AND X<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + W<63:0>);
V[d] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

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

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	20-16	15	14	13	12	11	10	9-5	4-0
1	1	0	0	1	1	1	0	0	1	1	Rm	1	0	0	0	1	0	Rn	Rd

Advanced SIMD

```
SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D
```

```
if !HaveSHA512Ext() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

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

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig1;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

sig1 = ROR(X<127:64>, 19) EOR ROR(X<127:64>, 61) EOR '000000':X<127:70>;
(X<127:64>,61) EOR '000000':X<127:70>;
Vtmp<127:64> = W<127:64> + sig1 + Y<127:64>;
sig1 = ROR(X<63:0>, 19) EOR ROR(X<63:0>, 61) EOR '000000':X<63:6>;
(X<63:0>,61) EOR '000000':X<63:6>;
Vtmp<63:0> = W<63:0> + sig1 + Y<63:0>;
V[d] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

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

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	0	1	1	1	0	0	1	1	Rm					1	1	0	0	0	0	Rn					Rd				

Advanced SIMD

```
SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
if !HaveSM3Ext() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

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

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;

result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):(Vm<127:96>,15):ROL(Vm<95:64>, 15):(Vm<95:64>
(Vm<63:32>,15));

for i = 0 to 3
    if i == 3 then
    if i == 3 then
        result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
(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(resul
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

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

SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

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

Advanced SIMD

```
SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
if !HaveSM3Ext() then UnallocatedEncoding();
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) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7):(Vm<127:96>,7):ROL(Vm<95:64>, 7):(Vm<95:64>,7):ROL(Vm<63:32>, 7
(Vm<31:0>,7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
(tmp<31:0>,15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR(tmp2,15) EOR ROL(tmp2, 23);
(tmp2,23);
result<127:96> = result<127:96> EOR tmp2;
V[d] = result;[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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	20-16	15	14-10	9-5	4-0
1	1	0	0	1	1	1	0	0	1	0	Rm	0	Ra	Rn	Rd

Advanced SIMD

```
SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S
```

```
if !HaveSM3Ext() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer a = UInt(Ra);
```

Assembler Symbols

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

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) Va = V[a];
Vd<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
(Vn<127:96>,12) + Vm<127:96> + Va<127:96>) , 7);
Vd<95:0> = Zeros();
V[d] = Vd;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	20-16	15	14	13-12	11	10	9-5	4-0
1	1	0	0	1	1	1	0	0	1	0	Rm	1	0	imm2	0	0	Rn	Rd

Advanced SIMD

```
SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

```
if !HaveSM3Ext() then UnallocatedEncoding();
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) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
[Vm,i,32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
(Vd<127:96>,12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
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>;
(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13-12	11	10	9-5	4-0
1	1	0	0	1	1	1	0	0	1	0	Rm	1	0	imm2	0	1	Rn	Rd

Advanced SIMD

```
SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

```
if !HaveSM3Ext() then UnallocatedEncoding();
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) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
[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>;
(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>;
(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	1	0	0	1	0	Rm					1	0	imm2		1	0	Rn					Rd				

Advanced SIMD

```
SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

```
if !HaveSM3Ext() then UnallocatedEncoding();
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(128) result;
bits(32) TT1;

Wj = Elem[Vm, i, 32];
[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>;
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>;
(Vd<95:64>,19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR(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.

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	0	1	0	Rm	1	0	imm2	1	1	Rn	Rd

Advanced SIMD

```
SM3TT2B <Vd>.S, <Vn>.S, <Vm>.S[<imm2>]
```

```
if !HaveSM3Ext() then UnallocatedEncoding();
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>;
[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>;
(Vd<95:64>,19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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

Advanced SIMD

```
SM4E <Vd>.4S, <Vn>.4S
```

```
if !HaveSM4Ext() then UnallocatedEncoding();
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;
integer index;

roundresult =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;
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval, i, 8] =[intval,i,8]  = Sbox(Elem[intval, i, 8]);
[intval,i,8]);

    intval = intval EOR ROL(intval, 2) EOR(intval,2) EOR ROL(intval, 10) EOR(intval,10) EOR ROL(intval,
    intval = intval EOR roundresult<31:0>;
(intval,24);
    intval = intval EOR roundresult<31:0>;

    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
1	1	0	0	1	1	1	0	0	1	1	Rm	1	1	0	0	1	0	Rn	Rd

Advanced SIMD

```
SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
if !HaveSM4Ext() then UnallocatedEncoding();
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;
integer index;

roundresult = V[n];
for index = 0 to 3
    const = Elem[Vm, index, 32];
[Vm,index,32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;

    for i = 0 to 3
        Elem[intval, i, 8] =[intval,i,8] = Sbox(Elem[intval, i, 8]);
[intval,i,8]);

    intval = intval EOR ROL(intval, 13) EOR(intval,13) EOR ROL(intval, 23);
    intval = intval EOR roundresult<31:0>;
(intval,23);
    intval = intval EOR roundresult<31:0>;

    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	20–16	15	14–10	9–5	4–0
1	0	0	1	1	0	1	1	0	0	1	Rm	0	Ra	Rn	Rd
								U				o0			

64-bit

```
SMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

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

Alias	Is preferred when
SMULL	`Ra == '11111'`

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.

31	30	29	28	27	26	25	24	23	22	21	20 … 5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16	0	0	0	1	1

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);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

SMLAL, SMLAL2 (by element)

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The SMLAL instruction extracts vector elements from the lower half of the first source register, while the SMLAL2 instruction extracts vector elements from the upper half of the first source register.

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

31	30	29	28	27	26	25	24	23-22	21	20	19-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	1	size	L	M	Rm	0	0	1	0	H	0	Rn	Rd
		U											o2						

Vector

```
SMLAL{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 UnallocatedEncoding();

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

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
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>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLAL` instruction extracts each source vector from the lower half of each source register, while the `SMLAL2` instruction extracts each source vector from the upper half of each source register.

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

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	0	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type

```
SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

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

if size == '11' then ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) 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>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23 22	21	20	19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	1	size	L	M	Rm	0	1	1	0	H	0	Rn	Rd
		U											o2						

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 UnallocatedEncoding();

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

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
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>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	20-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	1	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type

```
SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

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

if size == '11' then ReservedValue();
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":

Q	2
0	[absent]
1	[present]

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

<Ta> — Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) 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>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	20-16	15	14-10	9-5	4-0
1	0	0	1	1	0	1	1	0	0	1	Rm	1	Ra	Rn	Rd
								U				o0			

64-bit

```
SMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

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

Alias	Is preferred when
SMNEGL	`Ra == '11111'`

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
1	0	0	1	1	0	1	1	0	1	0	Rm	0	(1)	(1)	(1)	(1)	(1)	Rn	Rd
								U							Ra				

64-bit

```
SMULH <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);~~(Rm);~~
~~integer a = UInt(Ra);              // ignored by UMULH/SMULH~~
~~integer destsize = 64;~~
~~integer datasize = destsize;~~
~~boolean unsigned = (U == '1');~~
```

Assembler Symbols

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

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

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

Operation

```
bits(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	0	1	1	1	1	size	L	M	Rm	1	0	1	0	H	0	Rn	Rd
		U																	

Vector

```
SMULL{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 UnallocatedEncoding();

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”:

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
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>;    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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.

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

Three registers, not all the same type

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

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

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

boolean unsigned = (U == '1');
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

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>;[result, e, 2*esize] = (element1 * elem

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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

31	30	29	28	27	26	25	24	23-22	21	20	19-16	15	14	13	12	11	10	9-5	4-0
0	1	0	1	1	1	1	1	size	L	M	Rm	0	0	1	1	H	0	Rn	Rd
													o2						

Scalar

```
SQDMLAL <Va><d>, <Vb><n>, <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 UnallocatedEncoding();

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

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

Vector

31	30	29	28	27	26	25	24	23-22	21	20	19-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	1	size	L	M	Rm	0	0	1	1	H	0	Rn	Rd
													o2						

Vector

```
SQDMLAL{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 UnallocatedEncoding();

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”:

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

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

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
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);
(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQDMLAL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLAL2` instruction extracts each source vector from the upper half of each source register.

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

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	0	1	1	1	1	0	size	1	Rm	1	0	0	1	0	0	Rn	Rd
													o1					

Scalar

```
SQDMLAL <Va><d>, <Vb><n>, <Vb><m>
```

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

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

Vector

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	size	1	Rm	1	0	0	1	0	0	Rn	Rd
													o1					

Vector

```
SQDMLAL{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 ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<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”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

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

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<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

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
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, 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);
(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SQDMLSL, SQDMLSL2 (by element)

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQDMLSL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLSL2` instruction extracts vector elements from the upper half of the first source register.

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

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23-22	21	20	19-16	15	14	13	12	11	10	9-5	4-0
0	1	0	1	1	1	1	1	size	L	M	Rm	0	1	1	1	H	0	Rn	Rd
													o2						

Scalar

```
SQDMLSL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

```
integer 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 UnallocatedEncoding();

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

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

Vector

31	30	29	28	27	26	25	24	23-22	21	20	19-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	1	size	L	M	Rm	0	1	1	1	H	0	Rn	Rd
													o2						

Vector

```
SQDMLSL{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 UnallocatedEncoding();

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”:

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

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

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
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);
(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQDMLSL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLSL2` instruction extracts each source vector from the upper half of each source register.

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

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					1	0	1	1	0	0	Rn					Rd				
																		o1													

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 ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					1	0	1	1	0	0	Rn					Rd				
																		o1													

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 ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<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”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

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

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<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

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
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, 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);
(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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	1	size	L	M	Rm	1	0	1	1	H	0	Rn	Rd

Scalar

```
SQDMULL <Va><d>, <Vb><n>, <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 UnallocatedEncoding();

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	1	size	L	M	Rm	1	0	1	1	H	0	Rn	Rd

Vector

```
SQDMULL{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 UnallocatedEncoding();

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

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

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

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) 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);(2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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 *FPSR*.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	1	0	size	1	Rm	1	1	0	1	0	0	Rn	Rd

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 ReservedValue();
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	Rd

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 ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

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

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

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<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”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

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

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<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

```
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);(2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see *SHADD*.

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

31	30	29	28	27	26	25	24	23-22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	0	size	1	Rm	0	0	0	1	0	1	Rn	Rd
		U																

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 ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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 +

V[d] = result;
```

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB.

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

System

```
SSBB
```

```
~~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 UnallocatedEncoding();~~

~~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> == '00' then~~
~~            op = MemBarrierOp_SSBB;~~
~~        elsif CRm<3:2> == '01' then~~
~~            op = MemBarrierOp_PSSBB;~~
~~        else~~
~~            types = MBReqTypes_All;~~
~~            domain = MBReqDomain_FullSystem~~// Empty.~~;~~
```

Operation

```
~~case op of~~
~~    when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);~~
~~    when MemBarrierOp_DMBDataMemoryBarrier(domain, types);~~
~~    when MemBarrierOp_ISBInstructionSynchronizationBarrier();~~
~~    when MemBarrierOp_SSBB~~SpeculativeSynchronizationBarrierToVA~~();~~
~~    when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA~~();
```

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	x	x	1	x	size		Rn					Rt				
									L							opcode															

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
0	Q	0	0	1	1	0	0	1	0	0	Rm					x	x	1	x	size		Rn					Rt				
									L							opcode															

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

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

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

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

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

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

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

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

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

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

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

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

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

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

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

```
ST1 { <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”:

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

<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”:

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<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 = 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 UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory.

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

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	0	S	size		Rn					Rt				
									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>]
```

```
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	0	Rm					x	x	0	S	size		Rn					Rt				
									L	R						opcode															

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

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

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

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

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

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

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

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

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

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

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

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

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

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

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

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11 10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	size	Rn	Rt
									L							opcode						

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

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11 10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	0	0	1	0	0	Rm	1	0	0	0	size	Rn	Rt
									L			opcode						

Immediate offset (Rm == 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<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”:

Q	<imm>
0	#16
1	#32

<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 = 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 UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	0	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	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	1	Rm					x	x	0	S	size		Rn					Rt				
									L	R						opcode															

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

```
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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	size		Rn					Rt				
									L							opcode															

No offset

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm					0	1	0	0	size		Rn					Rt				
									L							opcode															

Immediate offset (Rm == 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<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”:

Q	<imm>
0	#24
1	#48

<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 = 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 UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	0	Rm					x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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	9-5	4-0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	size	Rn	Rt
									L							opcode						

No offset

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11-10	9-5	4-0
0	Q	0	0	1	1	0	0	1	0	0	Rm	0	0	0	0	size	Rn	Rt
									L			opcode						

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

<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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<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”:

Q	<imm>
0	#32
1	#64

<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 = 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 UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

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 *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset]() and [Post-index]()

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11 10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	1	S	size	Rn	Rt
									L	R							opcode					

8-bit (opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.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	1	Rm	x	x	1	S	size	Rn	Rt
									L	R			opcode					

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.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.
<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.

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 UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
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.

STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L				Rs			o0			Rt2												

32-bit (size == 10)

```
STLLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STLLR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 <<integer t2 = UInt(size);(Rt2); // ignored by load/store single register
integer s =UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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_LIMITEDORDEREDMemOp_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.

ISA_v84A_A64_xml_00bet7 (old) | **htmldiff from-ISA_v84A_A64_xml_00bet7** | **(new) ISA_v84A_A64_xml_00bet7_OPT**

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)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
STLLRB <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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 n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data =case memop of
    when MemOp_STORE
        data = X[t];
Mem[address, 1,[address, dbytes, acctype] = data;

    when
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtendAccType_LIMITEDORDEREDMemOp_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.

STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset
(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L				Rs			o0			Rt2												

No offset

```
STLLRH <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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, dbytes, acctype] = data;

    when
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtendAccType_LIMITEDORDEREDMemOp_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.

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*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9-5	4-0
1	x	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn	Rt
size									L				Rs			o0			Rt2				

32-bit (size == 10)

```
STLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STLR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 <<integer t2 = UInt(size);(Rt2); // ignored by load/store single register
integer s =UInt(Rs);    // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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_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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L				Rs			o0			Rt2												

No offset

```
STLRB <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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 n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data =case memop of
    when MemOp_STORE
        data = X[t];
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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*.

31	30	29	28	27	26	25	24	23	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	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L				Rs			o0			Rt2												

No offset

```
STLRH <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);(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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

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, 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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	20–12	11	10	9–5	4–0
1	x	0	1	1	0	0	1	0	0	0	imm9	0	0	Rn	Rt
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;~~(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 UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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
        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;
    else
        XAccType_ORDEREDMemOp_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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	0	0	1	1	0	0	1	0	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
STLURB <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);~~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 UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;
    else
        XAccType_ORDEREDMemOp_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.

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	20–12	11	10	9–5	4–0
0	1	0	1	1	0	0	1	0	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
STLURH <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);~~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 UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 2,[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;
    else
        XAccType_ORDEREDMemOp_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.

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*.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14-10	9-5	4-0
1	sz	0	0	1	0	0	0	0	0	1	Rs	1	Rt2	Rn	Rt
									L			o0			

32-bit (sz == 0)

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit (sz == 1)

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 <<(Rs);    // ignored by all loads and store-release AccType acctype = if o0 == '1'
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXP*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "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.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || (s == t2) thenif 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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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   UnallocatedEncoding();
        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
            bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1:el2 else el2:el1;
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[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,[address, dbytes, acctype] = data;
            status = AccType_ORDEREDExclusiveMonitorsStatus] = 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));
                X[t] = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] =[t] = 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
1	x	0	0	1	0	0	0	0	0	0	Rs	1	(1)	(1)	(1)	(1)	(1)	Rn	Rt
size									L			o0			Rt2				

32-bit (size == 10)

```
STLXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STLXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s =integer t2 = UInt(Rs);     // ignored by all loads and store-release

integer elsize = 8 <<(Rt2); // ignored by load/store single register
integer s = UInt(size);(Rs);    // ignored by all loads and store-releaseAccType acctype = if o0 == '1'
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXR*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<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.

Operation

```
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 s == t thenif 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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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   UnallocatedEncoding();
        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';
// 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[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,[address, dbytes, acctype] = data;
```

```
            status = AccType_ORDEREDExclusiveMonitorsStatus] = 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));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] =[t] = 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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	20–16	15	14	13	12	11	10	9–5	4–0
0	0	0	0	1	0	0	0	0	0	0	Rs	1	(1)	(1)	(1)	(1)	(1)	Rn	Rt
size									L			o0		Rt2					

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(Rt2); // ignored by load
integer s =UInt(Rs);    // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED 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 s == t thenif 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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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   UnallocatedEncoding();
        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(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) 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[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, 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_ORDEREDExclusiveMonitorsStatus] = 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));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] =[t] = 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	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	0	0	0	Rs					1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L							o0		Rt2													

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(Rt2); // ignored by load
integer s =UInt(Rs);    // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXRH*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t thenif 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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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   UnallocatedEncoding();
        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(16) 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';
// 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[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, 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_ORDEREDExclusiveMonitorsStatus] = 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));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] =[t] = 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

31 30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1	0	1	1	0	0	0	0	imm7	Rt2	Rn	Rt
								L				

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>}]
```

```
// Empty.~~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 variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then~~(Rt2); AccType acctype = AccType_VECSTREAM;~~
~~MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;~~
~~if opc == '11' then ~~UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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,[address + 0      , dbytes, acctype] = data1; AccType_VECSTREAMMem] = 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;
        V[t]  = data1;
        V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_VECSTREAMMem] = 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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair*.

31	30	29	28	27	26	25	24	23	22	21–15	14–10	9–5	4–0
x	0	1	0	1	0	0	0	0	0	imm7	Rt2	Rn	Rt
opc									L				

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(Rt2); AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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];
data2 =     if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
Mem[address, dbytes,[address + 0     , dbytes, acctype] = data1; AccType_STREAMMem] = 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;
        X[t]  = data1;
        X[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_STREAMMem] = 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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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

31 30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1	0	1	1	0	0	1	0	imm7	Rt2	Rn	Rt
								L				

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

31 30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1	0	1	1	0	1	1	0	imm7	Rt2	Rn	Rt
								L				

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

31 30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1	0	1	1	0	1	0	0	imm7	Rt2	Rn	Rt
								L				

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 = 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

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then(Rt2); AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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
        data1 = V[t];
data2 = V[t2];
Mem[address, dbytes,[address + 0      , dbytes, acctype] = data1; AccType_VECMem] = data1;[address + dby

    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;
        V[t]  = data1;
        VAccType_VECMem] = data2;
[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.

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				
opc									L																						

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```
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				
opc									L																						

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 10)

```
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				
opc									L																						

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
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

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then(Rt2); AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 thenif memop ==
    MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;          // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 thenif 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      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
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_NORMALMem] = 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;
            XAccType_NORMALMem] = data2;
[t2] = 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					Rt				
								opc																							

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 UnallocatedEncoding();
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					Rt				
								opc																							

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 UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
size	1	1	1	1	0	1	x	0	imm12	Rn	Rt
							opc				

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 UnallocatedEncoding();
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

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
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;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
1	x	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size								opc							

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>], #<simm>
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
1	x	1	1	1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
size								opc							

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21–10	9–5	4–0
1	x	1	1	1	0	0	1	0	0	imm12	Rn	Rt
size								opc				

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;(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
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
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
    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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 thenif n == 31 then
    if memop !=
    MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31 30	29	28	27	26	25	24	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	1	Rm	option	S	1	0	Rn	Rt
							opc									

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 UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding();     // 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 "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> For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

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”:

S	<amount>
0	#0
1	#2

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”:

S	<amount>
0	#0
1	#3

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”:

S	<amount>
0	#0
1	#4

Shared Decode

```
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

```
bits(64) offset = ExtendReg(m, extend_type, shift);
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;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8,[address, datasize DIV 8, acctype]; AccType_VECV];[t] = data

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        [] = address;
    else
        XVSP[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-13	12	11	10	9-5	4-0
1	x	1	1	1	0	0	0	0	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
~~boolean wback = FALSE;~~
~~boolean postindex = FALSE;~~
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding();     // 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”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<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”:

S	<amount>
0	#0
1	#2

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”:

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

integer datasize = 8 << scale;(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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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
        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;
    else
        XAccType_NORMALMemOp_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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn					Rt				
size								opc																							

Post-index

`STRB <Wt>, [<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	1	0	0	0	0	0	0	imm9									1	1	Rn					Rt				
size								opc																							

Pre-index

`STRB <Wt>, [<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	0	0	imm12												Rn					Rt				
size								opc																							

Unsigned offset

`STRB <Wt>, [<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 *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

```
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
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
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 =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      UnallocatedEncoding();
        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    UnallocatedEncoding();
        when Constraint_NOP      EndOfInstruction();

if n == 31 thenif n == 31 then
    if memop !=
    MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
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, 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-13	12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	0	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

Extended register (option != 011)

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' thenboolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding();     // sub-word 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.

<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”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
(m, extend_type, shift);
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;
    else
        XAccType_NORMALMemOp_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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn					Rt				
size								opc																							

Post-index

```
STRH <Wt>, [<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	1	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn					Rt				
size								opc																							

Pre-index

```
STRH <Wt>, [<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	1	1	1	1	0	0	1	0	0	imm12												Rn					Rt				
size								opc																							

Unsigned offset

```
STRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
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 *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

```
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
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
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
    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      UnallocatedEncoding();
        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    UnallocatedEncoding();
        when Constraint_NOP      EndOfInstruction();

if n == 31 thenif n == 31 then
    if memop !=
    MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

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
            data = X[t];
Mem[address, 2,[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31	30	29	28	27	26	25	24	23	22	21	20-16	15-13	12	11	10	9-5	4-0
0	1	1	1	1	0	0	0	0	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

32-bit

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then~~boolean wback = FALSE;~~
~~boolean postindex = FALSE;~~
~~integer scale = UInt(size);~~
~~if option<1> == '0' then ~~UnallocatedEncoding();     // 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;~~
```

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”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<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”:

S	<amount>
0	#0
1	#1

Shared Decode

```
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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 2,[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;
    else
        XAccType_NORMALMemOp_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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
1	x	1	1	1	0	0	0	0	0	0	imm9	1	0	Rn	Rt
size								opc							

32-bit (size == 10)

```
STTR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
STTR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    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
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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_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[address, datasize DIV 8, acctype] = 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.

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*.

31	30	29	28	27	26	25	24	23	22	21	20-12	11	10	9-5	4-0
0	0	1	1	1	0	0	0	0	0	0	imm9	1	0	Rn	Rt
size								opc							

Unscaled offset

```
STTRB <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);
AccType acctype = AccType_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    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~~
~~        UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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_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[address, 1, acctype] = 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.

STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	1	1	1	1	0	0	0	0	0	0	imm9	1	0	Rn	Rt
size								opc							

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);
AccType acctype = AccType_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    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
        UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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_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[address, 2, acctype] = 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 (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.

31	30	29	28	27	26	25	24	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	0	Rn					Rt				
								opc																							

8-bit (size == 00 && opc == 00)

```
STUR <Bt>, [<Xn|SP>{, #<simm>}]
```

16-bit (size == 01 && opc == 00)

```
STUR <Ht>, [<Xn|SP>{, #<simm>}]
```

32-bit (size == 10 && opc == 00)

```
STUR <St>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11 && opc == 00)

```
STUR <Dt>, [<Xn|SP>{, #<simm>}]
```

128-bit (size == 00 && opc == 10)

```
STUR <Qt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
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;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8,[address, datasize DIV 8, acctype]; AccType_VECV];[t] = data

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        [] = address;
    else
        XVSP[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.

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20-12	11	10	9-5	4-0
1	x	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size								opc							

32-bit (size == 10)

```
STUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
STUR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;(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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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
        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;
    else
        XAccType_NORMALMemOp_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.

STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	0	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
STURB <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);~~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 UnallocatedEncoding();~~
~~    else~~
~~        // sign-extending load~~
~~        memop = MemOp_LOAD;~~
~~        if size == '10' && opc<0> == '1' then UnallocatedEncoding();~~
~~        regsize = if opc<0> == '1' then 32 else 64;~~
~~        signed = TRUE;~~

~~integer datasize = 8 << scale;~~
```

Operation

```
bits(64) address;
bits(8) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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;
    else
        XAccType_NORMALMemOp_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*.

31	30	29	28	27	26	25	24	23	22	21	20–12	11	10	9–5	4–0
0	1	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
STURH <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);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 UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if n == 31 thenif 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      UnallocatedEncoding();
        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      UnallocatedEncoding();
        when Constraint_NOP        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, 2,[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;
    else
        XAccType_NORMALMemOp_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.

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*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	sz	0	0	1	0	0	0	0	0	1	Rs					0	Rt2					Rn					Rt				
									L							o0															

32-bit (sz == 0)

```
STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit (sz == 1)

```
STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);     // ignored by load/store single register
integer s = UInt(Rs);     // ignored by all loads and store-release

integer elsize = 32 <<(Rs);    // ignored by all loads and store-release AccType acctype = if o0 == '1'
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;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 *STXP*.

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.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || (s == 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      UnallocatedEncoding();~~
~~        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   UnallocatedEncoding();
        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   UnallocatedEncoding();
        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~~
~~            bits(datasize DIV 2) el1 = ~~X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1:el2 else el2:el1;
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[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,~~[address, dbytes, acctype] = data;~~
~~            status = ~~ AccType_ATOMIC~~ExclusiveMonitorsStatus~~] = 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));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] =[t] = 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	0	0	Rs					0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L							o0		Rt2													

32-bit (size == 10)

```
STXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s =integer t2 = UInt(Rs);     // ignored by all loads and store-release

integer elsize = 8 <<(Rt2); // ignored by load/store single register
integer s = UInt(size);(Rs);    // ignored by all loads and store-releaseAccType acctype = if o0 == '1'
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXR*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "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.

Operation

```
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 s == t thenif 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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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   UnallocatedEncoding();
        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';
// 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[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,[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));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] =[t] = 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	Rs					0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L							o0		Rt2													

No offset

```
STXRB <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(Rt2); // ignored by load
integer s =UInt(Rs);    // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXRB*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <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 s == t thenif 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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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   UnallocatedEncoding();
        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(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) 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[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, 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;
    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));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] =[t] = 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	1	0	0	1	0	0	0	0	0	0	Rs	0	(1)	(1)	(1)	(1)	(1)	Rn	Rt
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(Rt2); // ignored by load
integer s = UInt(Rs);    // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDERED 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.

Operation

```
bits(64) address;
bits(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t thenif 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      UnallocatedEncoding();
        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   UnallocatedEncoding();
        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   UnallocatedEncoding();
        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(16) 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';
// 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[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, 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>;
            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 = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] =[t] = 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

31	30	29	28	27	26	25	24	23	22	21	20–16	15–13	12–10	9–5	4–0
sf	1	0	0	1	0	1	1	0	0	1	Rm	option	imm3	Rn	Rd
	op	S													

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>}}
```

```
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');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then 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 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”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

```
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;

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 thenif 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23 22	21 … 10	9 … 5	4 … 0
sf	1	0	1	0	0	0	1	shift	imm12	Rn	Rd
	op	S									

32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12:(imm12 :Zeros(12), datasize);
    when '1x' 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”:

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

Operation

```
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);
(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 thenif 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register).

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	0	0	1	0	1	1	shift	0	Rm	imm6	Rn	Rd
	op	S											

32-bit (sf == 0)

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
~~boolean sub_op = (op == '1');~~
~~boolean setflags = (S == '1');~~

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

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”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<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

Alias	Is preferred when
NEG (shifted register)	Rn == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
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 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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).

31	30	29	28	27	26	25	24	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	1	0	1	1	0	0	1	Rm					option			imm3			Rn					Rd				
	op	S																													

32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;~~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 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 first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

Alias	Is preferred when
CMP (extended register)	Rd == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;

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;

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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).

31	30	29	28	27	26	25	24	23-22	21-10	9-5	4-0
sf	1	1	1	0	0	0	1	shift	imm12	Rn	Rd
	op	S									

32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
~~boolean sub_op = (op == '1');~~
~~boolean setflags = (S == '1');~~
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12:~~(imm12 :~~Zeros(12), datasize);
    when '1x' 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”:

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

Alias Conditions

Alias	Is preferred when
CMP (immediate)	Rd == '11111'

Operation

```
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);
    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;

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	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	1	0	1	1	shift		0	Rm					imm6						Rn					Rd				
	op	S																													

32-bit (sf == 0)

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
SUBS <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 ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

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”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<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

Alias	Is preferred when
CMP (shifted register)	Rd == '11111'
NEGS	Rn == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;

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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SVC

Supervisor Call causes an exception to be taken to EL1.

On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in *ESR_ELx*, using the EC value 0x15, and the value of the immediate argument.

31	30	29	28	27	26	25	24	23	22	21	20–5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16	0	0	0	0	1

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);~~
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

[ISA_v84A_A64_xml_00bet7 (old)] htmldiff from-ISA_v84A_A64_xml_00bet7 [(new) ISA_v84A_A64_xml_00bet7_OPT]

SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					1	0	0	0	0	0	Rn					Rt				
size																															

SWPAB (A == 1 && R == 0)

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

SWPALB (A == 1 && R == 1)

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

SWPB (A == 0 && R == 0)

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

SWPLB (A == 0 && R == 1)

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();

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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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(8) data;
bits(datasize) data;

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];
Mem[address, 1, stacctype] =[address, datasize DIV 8, stacctype] = X[s];

X[t] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer
(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					1	0	0	0	0	0	Rn					Rt				
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 UnallocatedEncoding();

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_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW 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 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];
Mem[address, 2, stacctype] =[address, datasize DIV 8, stacctype] = X[s];

X[t] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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, DC, 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	1	0	0	0	0	1	op1	CRn	CRm	op2	Rt
										L							

System

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 1;
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);(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

Alias	Is preferred when
AT	`CRn == '0111' && CRm == '100x' && SysOp(op1,'0111',CRm,op2) == Sys_AT`
DC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_DC`
IC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_IC`
TLBI	`CRn == '1000' && SysOp(op1,'1000',CRm,op2) == Sys_TLBI`

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

SYSL

System instruction with result. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3 2 1 0
1	1	0	1	0	1	0	1	0	0	1	0	1	op1	CRn	CRm	op2	Rt
										L							

System

```
SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 1;
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);(CRm);
boolean has_result = (L == '1');
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
if has_result then
    X[t] = AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    AArch64.SysInstr(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X(1, sys_op1, sys_crn, sys_crm, sys_c
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

TBL

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 *CPACR_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	20-16	15	14-13	12	11	10	9-5	4-0
0	Q	0	0	1	1	1	0	0	0	0	Rm	0	len	0	0	0	Rn	Rd
														op				

Two register table (len == 01)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table (len == 10)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table (len == 11)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table (len == 00)

```
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23 22 21 20 19	18 17 16 15 14 13 12 11 10 9 8 7 6 5	4 3 2 1 0
b5	0	1	1	0	1	1	1	b40	imm14	Rt
							op			

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”:

b5	<R>
0	W
1	X

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == op thenif operand<bit_pos> == bit_val then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	Rm					0	len		1	0	0	Rn					Rd				
																			op												

Two register table (len == 01)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table (len == 10)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table (len == 11)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table (len == 00)

```
TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// 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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

31	30	29	28	27	26	25	24	23 22 21 20 19	18 17 16 15 14 13 12 11 10 9 8 7 6 5	4 3 2 1 0
b5	0	1	1	0	1	1	0	b40	imm14	Rt
							op			

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”:

b5	<R>
0	W
1	X

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == op thenif operand<bit_pos> == bit_val then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

[ISA_v84A_A64_xml_00bet7 (old)] htmldiff from-ISA_v84A_A64_xml_00bet7 [(new) ISA_v84A_A64_xml_00bet7_OPT]

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)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	1	0	1	1	1	1	1
																				CRm				op2							

System

```
TSB CSYNC
```

```
SystemHintOpif !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() then;
    when '0011 xxx'
        SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    otherwise EndOfInstruction();();                                 // Instruction executes as NOP
```

Operation

```
TraceSynchronizationBarrier();case op of
    whenSystemHintOp_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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

    otherwise // do nothing
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:~~58~~45

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.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	1	1	1	1	Rn	Rd
		U													ac			

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 ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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>;(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register, while the UABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	0	1	0	0	Rn	Rd
		U											op					

Three registers, not all the same type

```
UABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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 the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	1	1	0	1	Rn	Rd
		U													ac			

Three registers of the same type

```
UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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>;(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

UABDL, UABDL2

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABDL` instruction extracts each source vector from the lower half of each source register, while the `UABDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23-22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	Q	1	0	1	1	1	0	size	1	Rm	0	1	1	1	0	0	Rn	Rd
		U											op					

Three registers, not all the same type

```
UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	1	1	0	1	0	Rn	Rd
		U														op						

Vector

```
UADALP <Vd>.<Ta>, <Vn>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
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”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
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>;    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	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 ReservedValue();
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”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
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>;    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | htmldiff from-ISA_v84A_A64_xml_00bet7 | (new) ISA_v84A_A64_xml_00bet7_OPT

UBFM

Unigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.
If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.
If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.
In both cases the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH.

31	30	29	28	27	26	25	24	23	22	21-16	15-10	9-5	4-0
sf	1	0	1	0	0	1	1	0	N	immr	imms	Rn	Rd
	opc												

32-bit (sf == 0 && N == 0)

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit (sf == 1 && N == 1)

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' thencase 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' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt();

R = (immr);
S = UInt(immr);
(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.
	For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

<imms> For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
LSL (immediate)	32-bit	`imms != '011111' && imms + 1 == immr`
LSL (immediate)	64-bit	`imms != '111111' && imms + 1 == immr`
LSR (immediate)	32-bit	`imms == '011111'`
LSR (immediate)	64-bit	`imms == '111111'`
UBFIZ		`UInt(imms) < UInt(immr)`
UBFX		`BFXPreferred(sf, opc<1>, imms, immr)`
UXTB		`immr == '000000' && imms == '000111'`
UXTH		`immr == '000000' && imms == '001111'`

Operation

```
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;
~~(src<S>) else dst;~~

// combine extension bits and result bits
X[d] = bot AND tmask;~~[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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19	18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	1	1	1	0	type	0	0 0	0 1 1	scale	Rn	Rd
										rmode	opcode			

32-bit to half-precision (sf == 0 && type == 11)
(ARMv8.2)

```
UCVTF <Hd>, <Wn>, #<fbits>
```

32-bit to single-precision (sf == 0 && type == 00)

```
UCVTF <Sd>, <Wn>, #<fbits>
```

32-bit to double-precision (sf == 0 && type == 01)

```
UCVTF <Dd>, <Wn>, #<fbits>
```

64-bit to half-precision (sf == 1 && type == 11)
(ARMv8.2)

```
UCVTF <Hd>, <Xn>, #<fbits>
```

64-bit to single-precision (sf == 1 && type == 00)

```
UCVTF <Sd>, <Xn>, #<fbits>
```

64-bit to double-precision (sf == 1 && type == 01)

```
UCVTF <Dd>, <Xn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

rounding =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
        UnallocatedEncoding(FPCR);();
```

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;
```


ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type		1	0	0	0	1	1	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

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'    when '10'
        if opcode<2:1>:rmode != '11 01' then
        UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        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 UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding(FPCR);();
```

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

```
CheckFPAdvSIMDEnabled64();

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 = V[n];
        intval = FPToFixedJS(fltval, FPCR, TRUE);
        X[d] = ZeroExtend[d] = fltval;(intval<31:0>, 64);
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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.

31	30	29	28	27	26	25	24	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	1	0	Rm					0	0	0	0	1	0	Rn					Rd				
																					o1										

32-bit (sf == 0)

```
UDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
UDIV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;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, TRUE)) / Real((operand1, unsigned)) / Real(Int(operanc

X[d] = result<datasize-1:0>;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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)

31	30	29	28	27	26	25	24	23 22	21	20	19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	1	1	1	0	H	0	Rn	Rd
		U																	

Vector

```
UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then ReservedValue();
boolean signed = (U == '0');
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

```
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]);
[operand1, 4 * e + i, esize DIV 4]); 
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
[operand2, 4 * index + i, esize DIV 4]);
        else 
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
[operand1, 4 * e + i, esize DIV 4]); 
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
[operand2, 4 * index + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

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					Rd				
		U																													

Three registers of the same type

```
UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then~~if size!= '10' then~~ ReservedValue();
boolean signed = (U == '0');
~~boolean signed = (U=='0');~~
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
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]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
[operand2, 4 * e + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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	20-16	15	14-10	9-5	4-0
1	0	0	1	1	0	1	1	1	0	1	Rm	0	Ra	Rn	Rd
								U				o0			

64-bit

```
UMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);~~(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

Alias	Is preferred when
UMULL	`Ra == '11111'`

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) *~~
~~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.

31	30	29	28	27	26	25	24	23 22	21	20	19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	1	size	L	M	Rm	0	0	1	0	H	0	Rn	Rd
		U											o2						

Vector

```
UMLAL{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 UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
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>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	0	0	0	0	Rn	Rd
		U											o1					

Three registers, not all the same type

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> — Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) 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>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) | **htmldiff from-ISA_v84A_A64_xml_00bet7** | **(new) ISA_v84A_A64_xml_00bet7_OPT**

UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLSL` instruction extracts vector elements from the lower half of the first source register, while the `UMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23-22	21	20	19-16	15	14	13	12	11	10	9-5	4-0
0	Q	1	0	1	1	1	1	size	L	M	Rm	0	1	1	0	H	0	Rn	Rd
		U											o2						

Vector

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 — Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
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>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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.

31	30	29	28	27	26	25	24	23-22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	Q	1	0	1	1	1	0	size	1	Rm	1	0	1	0	0	0	Rn	Rd
		U											o1					

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 ReservedValue();
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”:

Q	2
0	[absent]
1	[present]

<Vd> — Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> — Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> — Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> — Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> — Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) 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>;
    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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

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.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14-10	9-5	4-0
1	0	0	1	1	0	1	1	1	0	1	Rm	1	Ra	Rn	Rd
								U				o0			

64-bit

```
UMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);(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

Alias	Is preferred when
UMNEGL	`Ra == '11111'`

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) *
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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
1	0	0	1	1	0	1	1	1	1	0	Rm	0	(1)	(1)	(1)	(1)	(1)	Rn	Rd
								U						Ra					

64-bit

```
UMULH <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);~~(Rm);~~
~~integer a = UInt(Ra);            // ignored by UMULH/SMULH~~
~~integer destsize = 64;~~
~~integer datasize = destsize;~~
~~boolean unsigned = (U == '1');~~
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 =~~bits(datasize) operand1 =~~ X[n];
bits(64) operand2 =~~bits(datasize) operand2 =~~ X[m];

integer result;

result = Int(operand1, TRUE) *~~(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.

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMULL` instruction extracts vector elements from the lower half of the first source register, while the `UMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				1	0	1	0	H	0	Rn					Rd				
		U																													

Vector

```
UMULL{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 UnallocatedEncoding();

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”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
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>;    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.

31	30	29	28	27	26	25	24	23-22	21	20-16	15	14	13	12	11	10	9-5	4-0
0	Q	1	0	1	1	1	0	size	1	Rm	1	1	0	0	0	0	Rn	Rd
		U																

Three registers, not all the same type

```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

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>;[result, e, 2*esize] = (element1 * elem

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 v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see *UHADD*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	size	1	Rm	0	0	0	1	0	1	Rn	Rd
		U																

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 ReservedValue();
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”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(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 +

V[d] = result;
```

WFE

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 SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event*.

As described in *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*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	1	1	1	1
																				CRm				op2							

System

```
WFE
```

```
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 111'
        SEE "XPACLRI";
    when '0001 xxx'
        SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
    when '0010 000'
        if !HaveRASExt() then EndOfInstruction();                   // Instruction executes as NOP
        op = SystemHintOp_ESB;
    when '0010 001'
        if !HaveStatisticalProfiling() then EndOfInstruction();     // Instruction executes as NOP
        op = SystemHintOp_PSB;
    when '0010 010'
        if !HaveSelfHostedTrace() then EndOfInstruction();          // Instruction executes as NOP
        op = SystemHintOp_TSB;
    when '0010 100'
        op = SystemHintOp_CSDB;
    when '0011 xxx'
        SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    otherwise EndOfInstruction// Empty.();                           // Instruction executes as
```

Operation

```
ifcase 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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();();

    otherwise // do nothing
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

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*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1
																				CRm				op2							

System

```
WFI
```

```
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";
    otherwise EndOfInstruction// Empty.();                                  // Instruction executes as
```

Operation

```
if !case op of
    whenSystemHintOp_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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();();

    otherwise // do nothing
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:5845

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT

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)

31	30	29	28	27	26	25	24	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	1	1	0	0	0	0	0	1	0	1	0	0	0	D	1	1	1	1	1	Rd				
																								Rn							

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
    UnallocatedEncoding() then();

if n != 31 then
    UnallocatedEncoding();
```

System
(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1

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);
```

ISA_v84A_A64_xml_00bet7 (old)	htmldiff from-ISA_v84A_A64_xml_00bet7	(new) ISA_v84A_A64_xml_00bet7_OPT

YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see *The YIELD instruction*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
																				CRm				op2							

System

```
YIELD
```

```
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 111'
        SEE "XPACLRI";
    when '0001 xxx'
        SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
    when '0010 000'
        if !HaveRASExt() then EndOfInstruction();                  // Instruction executes as NOP
        op = SystemHintOp_ESB;
    when '0010 001'
        if !HaveStatisticalProfiling() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_PSB;
    when '0010 010'
        if !HaveSelfHostedTrace() then EndOfInstruction();         // Instruction executes as NOP
        op = SystemHintOp_TSB;
    when '0010 100'
        op = SystemHintOp_CSDB;
    when '0011 xxx'
        SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
    otherwise  EndOfInstruction// Empty.();                                  // Instruction executes as
```

Operation

```
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();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSBProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();();

    otherwise // do nothing
```

Internal version only: isa v29.05, AdvSIMD v26.0, pseudocode v35.3 ; Build timestamp: 2018-06-16T09:58~~45~~

ISA_v84A_A64_xml_00bet7 (old) htmldiff from-ISA_v84A_A64_xml_00bet7 (new) ISA_v84A_A64_xml_00bet7_OPT