Arm® A32/T32 Instruction Set Architecture
Armv8, for Armv8-A architecture profile
Web Address

http://www.arm.com
AArch32 -- Base Instructions (alphabetic order)

**ADC, ADCS (immediate)**: Add with Carry (immediate).

**ADC, ADCS (register)**: Add with Carry (register).

**ADC, ADCS (register-shifted register)**: Add with Carry (register-shifted register).

**ADD (immediate, to PC)**: Add to PC: an alias of ADR.

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

**ADD, ADDS (register)**: Add (register).

**ADD, ADDS (register-shifted register)**: Add (register-shifted register).

**ADD, ADDS (SP plus immediate)**: Add to SP (immediate).

**ADD, ADDS (SP plus register)**: Add to SP (register).

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

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

**AND, ANDS (register)**: Bitwise AND (register).

**AND, ANDS (register-shifted register)**: Bitwise AND (register-shifted register).

**ASR (immediate)**: Arithmetic Shift Right (immediate): an alias of MOV, MOVS (register).

**ASR (register)**: Arithmetic Shift Right (register): an alias of MOV, MOVS (register-shifted register).

**ASRS (immediate)**: Arithmetic Shift Right, setting flags (immediate): an alias of MOV, MOVS (register).

**ASRS (register)**: Arithmetic Shift Right, setting flags (register): an alias of MOV, MOVS (register-shifted register).

**B**: Branch.

**BFC**: Bit Field Clear.

**BFI**: Bit Field Insert.

**BIC, BICS (immediate)**: Bitwise Bit Clear (immediate).

**BIC, BICS (register)**: Bitwise Bit Clear (register).

**BIC, BICS (register-shifted register)**: Bitwise Bit Clear (register-shifted register).

**BKPT**: Breakpoint.

**BL, BLX (immediate)**: Branch with Link and optional Exchange (immediate).

**BLX (register)**: Branch with Link and Exchange (register).

**BX**: Branch and Exchange.

**BXJ**: Branch and Exchange, previously Branch and Exchange Jazelle.

**CBNZ, CBZ**: Compare and Branch on Nonzero or Zero.

**CLREX**: Clear-Exclusive.

**CLZ**: Count Leading Zeros.

**CMN (immediate)**: Compare Negative (immediate).

**CMN (register)**: Compare Negative (register).

**CMN (register-shifted register)**: Compare Negative (register-shifted register).
**CMP (immediate)**: Compare (immediate).

**CMP (register)**: Compare (register).

**CMP (register-shifted register)**: Compare (register-shifted register).

**CPS, CPSID, CPSIE**: Change PE State.

**CRC32**: CRC32.

**CRC32C**: CRC32C.

**CSDB**: Consumption of Speculative Data Barrier.

**DBG**: Debug hint.

**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.

**DSB**: Data Synchronization Barrier.

**EOR, EORS (immediate)**: Bitwise Exclusive OR (immediate).

**EOR, EORS (register)**: Bitwise Exclusive OR (register).

**EOR, EORS (register-shifted register)**: Bitwise Exclusive OR (register-shifted register).

**ERET**: Exception Return.

**ESB**: Error Synchronization Barrier.

**HLT**: Halting Breakpoint.

**HVC**: Hypervisor Call.

**ISB**: Instruction Synchronization Barrier.

**IT**: If-Then.

**LDA**: Load-Acquire Word.

**LDAB**: Load-Acquire Byte.

**LDAEX**: Load-Acquire Exclusive Word.

**LDAEXR**: Load-Acquire Exclusive Byte.

**LDAEXD**: Load-Acquire Exclusive Doubleword.

**LDAEXH**: Load-Acquire Exclusive Halfword.

**LDAH**: Load-Acquire Halfword.

**LDC (immediate)**: Load data to System register (immediate).

**LDC (literal)**: Load data to System register (literal).

**LDM (exception return)**: Load Multiple (exception return).

**LDM (User registers)**: Load Multiple (User registers).

**LDM, LDMIA, LDMFD**: Load Multiple (Increment After, Full Descending).

**LMDA, LDMFA**: Load Multiple Decrement After (Full Ascending).

**LDMDB, LDMEA**: Load Multiple Decrement Before (Empty Ascending).
LDMIB, LDMED: Load Multiple Increment Before (Empty Descending).
LDR (immediate): Load Register (immediate).
LDR (literal): Load Register (literal).
LDR (register): Load Register (register).
LDRB (immediate): Load Register Byte (immediate).
LDRB (literal): Load Register Byte (literal).
LDRB (register): Load Register Byte (register).
LDRBT: Load Register Byte Unprivileged.
LDRD (immediate): Load Register Dual (immediate).
LDRD (literal): Load Register Dual (literal).
LDRD (register): Load Register Dual (register).
LDREX: Load Register Exclusive.
LDREXB: Load Register Exclusive Byte.
LDREXD: Load Register Exclusive Doubleword.
LDREXH: Load Register Exclusive Halfword.
LDRH (immediate): Load Register Halfword (immediate).
LDRH (literal): Load Register Halfword (literal).
LDRH (register): Load Register Halfword (register).
LDRHT: Load Register Halfword Unprivileged.
LDRSB (immediate): Load Register Signed Byte (immediate).
LDRSB (literal): Load Register Signed Byte (literal).
LDRSB (register): Load Register Signed Byte (register).
LDRSBT: Load Register Signed Byte Unprivileged.
LDRSH (immediate): Load Register Signed Halfword (immediate).
LDRSH (literal): Load Register Signed Halfword (literal).
LDRSH (register): Load Register Signed Halfword (register).
LDRSHT: Load Register Signed Halfword Unprivileged.
LDRT: Load Register Unprivileged.
LSL (immediate): Logical Shift Left (immediate): an alias of MOV, MOVS (register).
LSLS (immediate): Logical Shift Left, setting flags (immediate): an alias of MOV, MOVS (register).
LSR (immediate): Logical Shift Right (immediate): an alias of MOV, MOVS (register).
LSRS (immediate): Logical Shift Right, setting flags (immediate): an alias of MOV, MOVS (register).
LSRS (register): Logical Shift Right, setting flags (register): an alias of MOV, MOVS (register-shifted register).
MCR: Move to System register from general-purpose register or execute a System instruction.
MCRR: Move to System register from two general-purpose registers.
MLA, MLAS: Multiply Accumulate.
MLS: Multiply and Subtract.
MOV, MOVS (immediate): Move (immediate).
MOV, MOVS (register): Move (register).
MOVT: Move Top.
MRC: Move to general-purpose register from System register.
MRRC: Move to two general-purpose registers from System register.
MRS: Move Special register to general-purpose register.
MRS (Banked register): Move Banked or Special register to general-purpose register.
MSR (Banked register): Move general-purpose register to Banked or Special register.
MSR (immediate): Move immediate value to Special register.
MSR (register): Move general-purpose register to Special register.
MUL, MULS: Multiply.
MVN, MVNS (immediate): Bitwise NOT (immediate).
MVN, MVNS (register): Bitwise NOT (register).
NOP: No Operation.
ORN, ORNS (immediate): Bitwise OR NOT (immediate).
ORN, ORNS (register): Bitwise OR NOT (register).
ORR, ORRS (immediate): Bitwise OR (immediate).
ORR, ORRS (register): Bitwise OR (register).
ORR, ORRS (register-shifted register): Bitwise OR (register-shifted register).
PKHBT, PKHTB: Pack Halfword.
PLD (literal): Preload Data (literal).
PLD, PLDW (immediate): Preload Data (immediate).
PLD, PLDW (register): Preload Data (register).
PLI (immediate, literal): Preload Instruction (immediate, literal).
PLI (register): Preload Instruction (register).
POP: Pop Multiple Registers from Stack.
POP (multiple registers): Pop Multiple Registers from Stack: an alias of LDM, LDMIA, LDMFD.
POP (single register): Pop Single Register from Stack: an alias of LDR (immediate).
PSSBB: Physical Speculative Store Bypass Barrier.
PUSH: Push Multiple Registers to Stack.
PUSH (multiple registers): Push multiple registers to Stack: an alias of STMDB, STMFD.
PUSH (single register): Push Single Register to Stack: an alias of STR (immediate).
QADD: Saturating Add.
QADD16: Saturating Add 16.
QADD8: Saturating Add 8.
QASX: Saturating Add and Subtract with Exchange.
QDADD: Saturating Double and Add.
QDUB: Saturating Double and Subtract.
QAX: Saturating Subtract and Add with Exchange.
QSUB: Saturating Subtract.
QSUB16: Saturating Subtract 16.
QSUB8: Saturating Subtract 8.
RBIT: Reverse Bits.
REV: Byte-Reverse Word.
REV16: Byte-Reverse Packed Halfword.
REVSH: Byte-Reverse Signed Halfword.
RFE, RFEDA, RFEDB, RFEIA, RFEIB: Return From Exception.
ROR (immediate): Rotate Right (immediate): an alias of MOV, MOVS (register).
RORS (immediate): Rotate Right, setting flags (immediate): an alias of MOV, MOVS (register).
RRX: Rotate Right with Extend: an alias of MOV, MOVS (register).
RRXS: Rotate Right with Extend, setting flags: an alias of MOV, MOVS (register).
RSB, RSBS (immediate): Reverse Subtract (immediate).
RSB, RSBS (register): Reverse Subtract (register).
RSC, RSCS (immediate): Reverse Subtract with Carry (immediate).
RSC, RSCS (register): Reverse Subtract with Carry (register).
RSC, RSCS (register-shifted register): Reverse Subtract (register-shifted register).
SADD16: Signed Add 16.
SADD8: Signed Add 8.
SASX: Signed Add and Subtract with Exchange.
SB: Speculation Barrier.
SBC, SB (immediate): Subtract with Carry (immediate).
SBC, SB (register): Subtract with Carry (register).
SBC, SB (register-shifted register): Subtract with Carry (register-shifted register).
**SBFX**: Signed Bit Field Extract.

**SDIV**: Signed Divide.

**SEL**: Select Bytes.

**SETEND**: Set Endianness.

**SETPN**: Set Privileged Access Never.

**SEV**: Send Event.

**SEVL**: Send Event Local.

**SHADD16**: Signed Halving Add 16.

**SHADD8**: Signed Halving Add 8.

**SHASX**: Signed Halving Add and Subtract with Exchange.

**SHSAX**: Signed Halving Subtract and Add with Exchange.

**SHSUB16**: Signed Halving Subtract 16.

**SHSUB8**: Signed Halving Subtract 8.

**SMC**: Secure Monitor Call.

**SMLABBB, SMLABBT, SMLATB, SMLATT**: Signed Multiply Accumulate (halfwords).

**SMLAD, SMLADX**: Signed Multiply Accumulate Dual.

**SMLAL, SMLALS**: Signed Multiply Accumulate Long.

**SMLALBB, SMLALBT, SMLALTB, SMLALTT**: Signed Multiply Accumulate Long (halfwords).

**SMLALD, SMLALDX**: Signed Multiply Accumulate Long Dual.

**SMLAWB, SMLAWT**: Signed Multiply Accumulate (word by halfword).

**SMLSD, SMLSDX**: Signed Multiply Subtract Dual.

**SMLSDL, SMLSDLX**: Signed Multiply Subtract Long Dual.

**SMMLA, SMMLAR**: Signed Most Significant Word Multiply Accumulate.

**SMMLS, SMMLSR**: Signed Most Significant Word Multiply Subtract.

**SMMUL, SMMULR**: Signed Most Significant Word Multiply.

**SMUAD, SMUADX**: Signed Dual Multiply Add.

**SMULBB, SMULBT, SMULTB, SMULTT**: Signed Multiply (halfwords).

**SMULL, SMULLS**: Signed Multiply Long.

**SMULWB, SMULWT**: Signed Multiply (word by halfword).

**SMUSD, SMUSDX**: Signed Multiply Subtract Dual.

**SRS, SRSDA, SRSDB, SRSIA, SRSIB**: Store Return State.

**SSAT**: Signed Saturate.

**SSAT16**: Signed Saturate 16.

**SSAX**: Signed Subtract and Add with Exchange.

**SSBB**: Speculative Store Bypass Barrier.

**SSUB16**: Signed Subtract 16.
SSUB8: Signed Subtract 8.
STC: Store data to System register.
STL: Store-Release Word.
STLB: Store-Release Byte.
STLEX: Store-Release Exclusive Word.
STLEXB: Store-Release Exclusive Byte.
STLEXD: Store-Release Exclusive Doubleword.
STLEXH: Store-Release Exclusive Halfword.
STLH: Store-Release Halfword.
STM (User registers): Store Multiple (User registers).
STM, STMIA, STMEA: Store Multiple (Increment After, Empty Ascending).
STMDB, STMFD: Store Multiple Decrement Before (Full Descending).
STMIB, STMFA: Store Multiple Increment Before (Full Ascending).
STR (immediate): Store Register (immediate).
STR (register): Store Register (register).
STRB (immediate): Store Register Byte (immediate).
STRB (register): Store Register Byte (register).
STRBT: Store Register Byte Unprivileged.
STRD (immediate): Store Register Dual (immediate).
STRD (register): Store Register Dual (register).
STREX: Store Register Exclusive.
STREXB: Store Register Exclusive Byte.
STREXD: Store Register Exclusive Doubleword.
STREXH: Store Register Exclusive Halfword.
STRH (immediate): Store Register Halfword (immediate).
STRH (register): Store Register Halfword (register).
STRHT: Store Register Halfword Unprivileged.
STRT: Store Register Unprivileged.
SUB (immediate, from PC): Subtract from PC: an alias of ADR.
SUB, SUBS (immediate): Subtract (immediate).
SUB, SUBS (register): Subtract (register).
SUB, SUBS (SP minus immediate): Subtract from SP (immediate).
SUB, SUBS (SP minus register): Subtract from SP (register).
SVC: Supervisor Call.
SXTAB: Signed Extend and Add Byte.
SXTAB16: Signed Extend and Add Byte 16.
SXTAH: Signed Extend and Add Halfword.
SXTB: Signed Extend Byte.
SXTB16: Signed Extend Byte 16.
SXTH: Signed Extend Halfword.
TBB, TBH: Table Branch Byte or Halfword.
TEQ (immediate): Test Equivalence (immediate).
TEQ (register): Test Equivalence (register).
TEQ (register-shifted register): Test Equivalence (register-shifted register).
TSB CSYNC: Trace Synchronization Barrier.
TST (immediate): Test (immediate).
TST (register): Test (register).
TST (register-shifted register): Test (register-shifted register).
UADD16: Unsigned Add 16.
UADDB: Unsigned Add 8.
UASX: Unsigned Add and Subtract with Exchange.
UBFX: Unsigned Bit Field Extract.
UDF: Permanently Undefined.
UDIV: Unsigned Divide.
UHADD16: Unsigned Halving Add 16.
UHADD8: Unsigned Halving Add 8.
UHASX: Unsigned Halving Add and Subtract with Exchange.
UHSAAX: Unsigned Halving Subtract and Add with Exchange.
UHSUB16: Unsigned Halving Subtract 16.
UHSUB8: Unsigned Halving Subtract 8.
UMAAL: Unsigned Multiply Accumulate Accumulate Long.
UMLAL, UMLALS: Unsigned Multiply Accumulate Long.
UMULL, UMULLS: Unsigned Multiply Long.
UQADD16: Unsigned Saturating Add 16.
UQADD8: Unsigned Saturating Add 8.
UQASX: Unsigned Saturating Add and Subtract with Exchange.
UQSAX: Unsigned Saturating Subtract and Add with Exchange.
UQSUB16: Unsigned Saturating Subtract 16.
UQSUB8: Unsigned Saturating Subtract 8.
USAD8: Unsigned Sum of Absolute Differences.
USADA8: Unsigned Sum of Absolute Differences and Accumulate.
USAT: Unsigned Saturate.
USAT16: Unsigned Saturate 16.
USAX: Unsigned Subtract and Add with Exchange.
USUB16: Unsigned Subtract 16.
USUB8: Unsigned Subtract 8.
UXTAB: Unsigned Extend and Add Byte.
UXTAB16: Unsigned Extend and Add Byte 16.
UXTAH: Unsigned Extend and Add Halfword.
UXTB: Unsigned Extend Byte.
UXTB16: Unsigned Extend Byte 16.
UXTH: Unsigned Extend Halfword.
WFE: Wait For Event.
WFI: Wait For Interrupt.
YIELD: Yield hint.

Internal version only: isa v01_19, pseudocode v2020-09.xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADC, ADCS (immediate)

Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR <current_mode>.
  - The PE checks SPSR <current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ![1111] | 0 | 0 | 1 | 0 | 1 | 0 | 1 | S | Rn | Rd | imm12 |
| cond |

ADC (S == 0)

ADC{<c>}{<q>} \{<Rd>,\} \{<Rn>, \#<const>

ADCS (S == 1)

ADCS{<c>}{<q>} \{<Rd>,\} \{<Rn>, \#<const> |

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{setflags} = (S == '1'); \quad \text{imm32} = \text{A32ExpandImm}(\text{imm12});
\]

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | i | 0 | 1 | 0 | 1 | 0 | S | Rn | 0 | imm3 | Rd | imm8 |

ADC (S == 0)

ADC{<c>}{<q>} \{<Rd>,\} \{<Rn>, \#<const>

ADCS (S == 1)

ADCS{<c>}{<q>} \{<Rd>,\} \{<Rn>, \#<const> |

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{setflags} = (S == '1'); \quad \text{imm32} = \text{T32ExpandImm}(\text{i:imm3:imm8});
\]

if \(d == 15 \) || \(n == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
  • For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  • For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], imm32, PSTATE.C);
    if d == 15 then  // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
  • The execution time of this instruction is independent of:
      ◦ The values of the data supplied in any of its registers.
      ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
      ◦ The values of the data supplied in any of its registers.
      ◦ The values of the NZCV flags.
ADC, ADCS (register)

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

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|----------------|----------------|----------------|----------------|
| != 1111 | 0 0 0 0 | 1 0 1 | S | Rn | Rd | imm5 | stype | 0 | Rm |

cond

ADC, rotate right with extend (S == 0 & imm5 == 00000 & stype == 11)

ADC{<c}>{<q>} {<Rd>,} <Rn>, <Rm>, RRX

ADC, shift or rotate by value (S == 0 & ! (imm5 == 00000 & stype == 11))

ADC{<c}>{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

ADCS, rotate right with extend (S == 1 & imm5 == 00000 & stype == 11)

ADCS{<c}>{<q>} {<Rd>,} <Rn>, <Rm>, RRX

ADCS, shift or rotate by value (S == 1 & ! (imm5 == 00000 & stype == 11))

ADCS{<c}>{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|
| 0 1 0 0 0 0 | 0 1 0 1 | Rm | Rdn |

T1

ADC{<c>} {<Rd>,} <Rdn>, <Rm> // (Inside IT block)

ADCS{<c>} {<Rd>,} <Rdn>, <Rm> // (Outside IT block)

d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = ! InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);

T2
ADC, rotate right with extend (S == 0 & imm3 == 000 & imm2 == 00 & stype == 11)

ADC{<c>{<q>}{<Rd>,}<Rn>,<Rm>}, RRX

ADC, shift or rotate by value (S == 0 & !(imm3 == 000 & imm2 == 00 & stype == 11))

ADC{<c>.W{<Rd>,}<Rn>,<Rm} // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
ADC{<c>{<q>{<Rd>,}<Rn},<Rm} {,<shift>#<amount>}

ADCS, rotate right with extend (S == 1 & imm3 == 000 & imm2 == 00 & stype == 11)

ADCS{<c>{<q>}{<Rd>,}<Rn>,<Rm} , RRX

ADCS, shift or rotate by value (S == 1 & !(imm3 == 000 & imm2 == 00 & stype == 11))

ADCS.W{<Rd>,} <Rn>, <Rm} // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
ADCS{<c>{<q>}{<Rd>,}<Rn},<Rm} {,<shift>#<amount>}

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

• For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
• For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

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

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.
For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

In T32 assembly:
- Outside an IT block, if ADCS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADCS <Rd>, <Rn> had been written.
- Inside an IT block, if ADC<e> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADC<e> <Rd>, <Rn> had been written.
To prevent either of these happening, use the .W qualifier.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADC, ADCS (register-shifted register)

Add with Carry (register-shifted register) adds a register value, the Carry flag value, and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 1111 0 0 0 0 1 0 1 S Rn Rd Rs 0 stype 1 Rm</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

Flag setting (S == 1)

ADCs{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

Not flag setting (S == 0)

ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<st> See Standard assembler syntax fields.
<st> See Standard assembler syntax fields.
<st> Is the general-purpose destination register, encoded in the "Rd" field.
<st> Is the first general-purpose source register, encoded in the "Rn" field.
<st> Is the second general-purpose source register, encoded in the "Rm" field.
<st> Is the type of shift to be applied to the second source register, encoded in "stype":

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

<st> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(Rs[7:0]);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
ADD (immediate, to PC)

Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This is a pseudo-instruction of ADR. This means:

- The encodings in this description are named to match the encodings of ADR.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ADR gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T3).

### A1

![A1 encoding]

\[ ADD{<c>}{<q>} <Rd>, PC, #<const> \]

is equivalent to

\[ ADR{<c>}{<q>} <Rd>, <label> \]

### T1

![T1 encoding]

\[ ADD{<c>}{<q>} <Rd>, PC, #<imm8> \]

is equivalent to

\[ ADR{<c>}{<q>} <Rd>, <label> \]

### T3

![T3 encoding]

\[ ADDW{<c>}{<q>} <Rd>, PC, #<imm12> // (<Rd>, <imm12> can be represented in T1) \]

\[ ADD{<c>}{<q>} <Rd>, PC, #<imm12> \]

is equivalent to

\[ ADR{<c>}{<q>} <Rd>, <label> \]

**Assembler Symbols**

- `<c>`  See *Standard assembler syntax fields*.
- `<q>`  See *Standard assembler syntax fields*. 
<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If the PC is is the branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For encoding T1 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<label> For encoding A1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.

If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset.
If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32.
Permitted values of the size of the offset are any of the constants described in Modified immediate constants in A32 instructions.

For encoding T1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For encoding T3: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.

If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.
If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32.
Permitted values of the size of the offset are 0-4095.

<imm8> Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> An immediate value. See Modified immediate constants in A32 instructions for the range of values.

Operation

The description of ADR gives the operational pseudocode for this instruction.
ADD, ADDS (immediate)

Add (immediate) adds an immediate value to a register value, and writes the result to the destination register. If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3 and T4).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------|-----------------------------|-----------------------------|
| != 1111                    | 0 0 1 0 1 0 0  S            | Rn                     |
| cond                       | Rd                           | imm12                     |

ADD (S == 0 && Rn != 11x1)

ADD{s}{c}<q> {<Rd>,} <Rn>, #<const>

ADDS (S == 1 && Rn != 1101)

ADDS{s}{c}<q> {<Rd>,} <Rn>, #<const>

if Rn == '1111' && S == '0' then SEE "ADR";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);

T1

| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------|-----------------------------|-----------------------------|
| 0 0 1 1 1 0  imm3           | Rn                         |
| Rd                          | imm32                      |

T1

ADD<s>{c} <Rd>, <Rn>, #<imm3> // (Inside IT block)

ADDS<s> <Rd>, <Rn>, #<imm3> // (Outside IT block)

d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);

T2

| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------|-----------------------------|-----------------------------|
| 0 0 1 1 0  Rdn              | imm8                        |
ADD\(<c>\{<q>\} <Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> can be represented in T1)
ADD\(<c>\{<q>\} \{<Rdn>, <imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)
ADD\(S\{<c>\{<q>\} \{<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)
ADD\(S\{<c>\{<q>\} \{<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)
\[
d = \text{UInt}(Rdn); \ n = \text{UInt}(Rdn); \ \text{setflags} = \text{InITBlock}(); \ \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32);
\]
ADD\(W\{<c>\{<q>\} \{<Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> can be represented in T1 or T2)
ADD\(W\{<c>\{<q>\} \{<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1 or T2)
\[
\text{imm32} = \text{T32ExpandImm}(i:\text{imm3}:\text{imm8});
\]
ADDWS{<c>\{<q>\} \{<Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)
ADDWS{<c>\{<q>\} \{<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)
\[
d = \text{UInt}(Rdn); \ n = \text{UInt}(Rdn); \ \text{setflags} = \text{InITBlock}(); \ \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32);
\]
ADDW{<c>\{<q>\} \{<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)
\[
\text{imm32} = \text{T32ExpandImm}(i:\text{imm3}:\text{imm8});
\]
if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
if Rn == '11101' then SEE "ADD (SP plus immediate)";
if d == 15 then UNPREDICTABLE;   // Armv8-A removes UNPREDICTABLE for R13
For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<Rdn>\) Is the general-purpose source and destination register, encoded in the "Rdn" field.
\(<\text{imm8}>\) Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ADDS variant, the instruction performs an exception return, that restores \texttt{PSTATE} from SPSR_\texttt{<current\_mode>}. Arm deprecates use of this instruction.

For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see \texttt{ADD (SP plus immediate)}. If the PC is used, see \texttt{ADR}.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see \texttt{ADD (SP plus immediate)}.

<imm3> is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

<imm12> is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T3: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the ADDW syntax). Encoding T1 is preferred to encoding T2 if <Rd> is specified and encoding T2 is preferred to encoding T1 if <Rd> is omitted.

**Operation**

```plaintext
if \texttt{CurrentInstrSet()} == \texttt{InstrSet\_A32} then
  if \texttt{ConditionPassed()} then
    EncodingSpecificOperations();
    (result, nzcv) = \texttt{AddWithCarry(R[n], imm32, '0')};
    if d == 15 then          // Can only occur for A32 encoding
      if setflags then
        \texttt{ALUExceptionReturn(result)};
      else
        \texttt{ALUWritePC(result)};
    else
      \texttt{R[d] = result};
      if setflags then
        \texttt{PSTATE.<N,Z,C,V> = nzcv};
  else
    if \texttt{ConditionPassed()} then
      EncodingSpecificOperations();
      (result, nzcv) = \texttt{AddWithCarry(R[n], imm32, '0')};
      \texttt{R[d] = result};
      if setflags then
        \texttt{PSTATE.<N,Z,C,V> = nzcv};
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Add (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register. If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

• The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
• The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  ◦ The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  ◦ The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  ◦ The instruction is UNDEFINED in Hyp mode.
  ◦ The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

A1

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 5 | != 1101 | Rd | imm5 | stype | 0 | Rm |
```

ADD, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
ADD{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, RRX
```

ADD, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
ADD{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ADDS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
ADDS{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, RRX
```

ADDS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
ADDS{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

if Rn == '1101' then SEE "ADD (SP plus register)"

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');  
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | Rd | Rm | Rn |
```

T1

```
ADD{<c>}{<q>}{<Rd>,} <Rn>, <Rm> // (Inside IT block)
```

```
ADDS{<c>}{<Rd>,} <Rm> // (Outside IT block)
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();  
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```
T2

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 DN != 1101 Rdn

T2 (!DN == 1 && Rd == 101)

ADD<c>{<q>} {<Rdn>, <Rm>} // (Preferred syntax, Inside IT block)

ADD<c>{<q>} {<Rdn>, <Rm>}

if (DN:Rdn) == '1101' || Rd == '1101' then SEE "ADD (SP plus register)";
d = UAInt(DN:Rdn); n = d; m = UAInt(Rm); setflags = FALSE; (shift_t, shift_n) = (SRTYPE_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;

T3

ADD, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

ADD<c>{<q>} {<Rd>,} <Rn>, <Rm>, RRX

ADD, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ADD<c>.W {<Rd>,} <Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
ADD<c>.W {<Rd>,} <Rn>, <Rm> // (Preferred syntax, Inside IT block)
ADD<c>{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

ADDS, rotate right with extend (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

ADDS<c>{<q>} {<Rd>,} <Rn>, <Rm>, RRX

ADDS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

ADDS.W {<Rd>,} <Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2)
ADDS<c>{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

if Rd == '1111' && S == '1' then SEE "CMN (register)";
if Rn == '1101' then SEE "ADD (SP plus register)";
d = UAInt(Rd); n = UAInt(Rn); m = UAInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.

ADD, ADDS (register)
<Rdn> is the general-purpose source and destination register, encoded in the "DN:Rdn" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. The assembler language allows <Rdn> to be specified once or twice in the assembler syntax. When used inside an IT block, and <Rdn> and <Rm> are in the range R0 to R7, <Rdn> must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when <Rdn> is specified once or twice.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR.<current_mode>. Arm deprecates use of this instruction.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. When used inside an IT block, <Rd> must be specified. When used outside an IT block, <Rd> is optional, and:

- If omitted, this register is the same as <Rn>.
- If present, encoding T1 is preferred to encoding T2.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used. If the SP is used, see *ADD (SP plus register)*.

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see *ADD (SP plus register)*.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T3: is the second general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

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

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Inside an IT block, if ADD<c> <Rd>, <Rn>, <Rd> cannot be assembled using encoding T1, it is assembled using encoding T2 as though ADD<c> <Rd>, <Rn> had been written. To prevent this happening, use the .W qualifier.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end if
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
        end if
    end if
```
Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

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

A1

Flag setting (S == 1)

ADD{<c}>{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

Not flag setting (S == 0)

ADD{<c}>{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONstrained UNPREDICTable behavior, see Architectural Constraints on
UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.

< Rd > Is the general-purpose destination register, encoded in the "Rd" field.

< Rn > Is the first general-purpose source register, encoded in the "Rn" field.

< Rm > Is the second general-purpose source register, encoded in the "Rm" field.

< shift > Is the type of shift to be applied to the second source register, encoded in "stype":

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

< Rs > Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the
"Rs" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or
destination:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
ADD, ADDS (SP plus immediate)

Add to SP (immediate) adds an immediate value to the SP value, and writes the result to the destination register. If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:
- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3 and T4).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| != 1111 | 0 0 1 0 | 1 0 0 | 5 | 1 1 0 1 | Rd | imm12 |
```

ADD (S == 0)

```
ADD{<c>}{<q>} {<Rd>,} SP, #<const>
```

ADDS (S == 1)

```
ADDS{<c>}{<q>} {<Rd>,} SP, #<const>
```

```
d = UInt(Rd); setflags = (S == '1'); imm32 = A32ExpandImm(imm12);
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 0 1 0 | 1 | Rd | imm8 |
```

T1

```
ADD{<c>}{<q>} <Rd>, SP, #<imm8>
```

```
d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(imm8:'00', 32);
```

T2

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 0 1 1 0 0 0 0 | 0 | imm7 |
```

T2

```
ADD{<c>}{<q>} {SP,} SP, #<imm7>
```

```
d = 13; setflags = FALSE; imm32 = ZeroExtend(imm7:'00', 32);
```

T3
ADD (S == 0)

ADD{<c>}.W {<Rd>}, SP, #<const> // (Rd, <const> can be represented in T1 or T2)

ADD{<c>}{<q>} {<Rd>}, SP, #<const>

ADDS (S == 1 & Rd != 1111)

ADDS{<c>}{<q>} {<Rd>}, SP, #<const>

if Rd == '1111' & S == '1' then SEE "CMN (immediate)"

d = UInt(Rd); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 & !setflags then UNPREDICTABLE;

T4

ADD{<c>}{<q>} {<Rd>}, SP, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)

ADDW{<c>}{<q>} {<Rd>}, SP, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>`: See Standard assembler syntax fields.
- `<q>`: See Standard assembler syntax fields.
- SP: Is the stack pointer.
- `<imm7>`: Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as `<imm7>/4`.
- `<Rd>`: For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  - For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
  For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.
  For encoding T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.
- `<imm8>`: Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as `<imm8>/4`.
- `<imm12>`: Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
- `<const>`: For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.
  For encoding T3: an immediate value. See Modified immediate constants in T32 instructions for the range of values.
Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  (result, nzcv) = AddWithCarry(SP, imm32, '0');
  if d == 15 then          // Can only occur for A32 encoding
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
  else
    R[d] = result;
    if setflags then
      PSTATE.<N,Z,C,V> = nzcv;
ADD, ADDS (SP plus register)

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

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ADDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

cond

ADD, rotate right with extend (S == 0 & imm5 == 00000 & stype == 11)

ADD{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX

ADD, shift or rotate by value (S == 0 & !(imm5 == 00000 & stype == 11))

ADD{<c>}{<q>} {<Rd>,} SP, <Rm> {}, shift #<amount>

ADDS, rotate right with extend (S == 1 & imm5 == 00000 & stype == 11)

ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX

ADDS, shift or rotate by value (S == 1 & !(imm5 == 00000 & stype == 11))

ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> {}, shift #<amount>

d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

T1

ADD{<c>}{<q>} {<Rdm>,} SP, <Rdm>

d = UInt(DM:Rdm); m = UInt(DM:Rdm); setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 & InITBlock() & !LastInITBlock() then UNPREDICTABLE;
ADD\{<c}\}{<q}\} \{SP,} SP, <Rm>

if Rm == '1101' then SEE "encoding T1";
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, 0);

ADD, rotate right with extend (S == 0 & \& imm3 == 000 & \& imm2 == 00 & \& stype == 11)

ADD\{<c}\}{<q}\} \{<Rd>,} SP, <Rm>, RRX

ADD, shift or rotate by value (S == 0 & \& !(imm3 == 000 & \& imm2 == 00 & \& stype == 11))

ADD\{<c}\}.W \{<Rd>,} SP, <Rm> // (<Rd>, <Rm> can be represented in T1 or T2)
ADD\{<c}\}{<q}\} \{<Rd>,} SP, <Rm> \{, <shift> \#<amount>\}

For more information about the CONstrained UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.
- For the ADDS variant, the instruction performs an exception return, that restores `PSTATE` from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<Rm>

For encoding A1 and T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the second general-purpose source register, encoded in the "Rm" field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

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

<amount>

For encoding A1: is the shift amount, in the range 1 to 31 (when `<shift>` = LSL or ROR) or 1 to 32 (when `<shift>` = LSR or ASR) encoded in the “imm5” field as `<amount>` modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when `<shift>` = LSL or ROR) or 1 to 32 (when `<shift>` = LSR or ASR), encoded in the “imm3:imm2” field as `<amount>` modulo 32.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(SP, shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

This instruction is used by the alias **SUB (immediate, from PC)**.

This instruction is used by the pseudo-instruction **ADD (immediate, to PC)**.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1, T2 and T3).

### A1

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|-----------|----------------|
| != 1111 | 0 0 1 0 | 0 1 0 | 0 | 1 1 1 1 | Rd | imm12 |
| cond     |          |      |       |         |    |       |
```

**A1**

```plaintext```
ADR{<c>}{<q>}  <Rd>,  <label>

d = UInt(Rd);  imm32 = A32ExpandImm(imm12);  add = TRUE;
```

### A2

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|-----------|----------------|
| != 1111 | 0 0 1 0 | 0 1 0 | 0 | 1 1 1 1 | Rd | imm12 |
| cond     |          |      |       |         |    |       |
```

**A2**

```plaintext```
ADR{<c>}{<q>}  <Rd>,  <label>

d = UInt(Rd);  imm32 = A32ExpandImm(imm12);  add = FALSE;
```

### T1

```
| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|-----------|
| 1 0 1 0 | 0 | Rd | imm8 |
```

**T1**

```plaintext```
ADR{<c>}{<q>}  <Rd>,  <label>

d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

### T2

```
| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|-----------|
| 1 1 1 1 0 | i | 1 0 1 | 0 | 1 0 1 | 1 1 1 1 0 | imm3 | Rd | imm8 |
```

**T2**

```plaintext```
ADR{<c>}{<q>}  <Rd>,  <label>

d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = FALSE;
if d == 15 then UNPREDICTABLE;        // Armv8-A removes UNPREDICTABLE for R13
### Assembler Symbols

- **<c>** See *Standard assembler syntax fields*.
- **<q>** See *Standard assembler syntax fields*.
- **<Rd>** For encoding A1 and A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
  - For encoding T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
- **<label>** For encoding A1 and A2: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.
  - If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset.
  - If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32.
  - Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions*.
  - For encoding T1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.
  - For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.
  - If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.
  - If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32.
  - Permitted values of the size of the offset are 0-4095.

The instruction aliases permit the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD (immediate, to PC)</strong></td>
<td></td>
<td>Never</td>
</tr>
<tr>
<td><strong>SUB (immediate, from PC)</strong></td>
<td>T2</td>
<td>i:imm3:imm8 == '000000000000'</td>
</tr>
<tr>
<td><strong>SUB (immediate, from PC)</strong></td>
<td>A2</td>
<td>imm12 == '000000000000'</td>
</tr>
</tbody>
</table>
if `ConditionPassed`() then
    EncodingSpecificOperations();
    result = if add then (Align(PC, 4) + imm32) else (Align(PC, 4) - imm32);
    if d == 15 then  // Can only occur for A32 encodings
        ALUWritePC(result);
    else
        R[d] = result;
AND, ANDS (immediate)

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

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccc}
\hline
! &= 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & S & Rn & Rd & \text{imm12} \\
\text{cond} & \hline
\end{array}
\]

**AND (S == 0)**

AND\{<c>\}{<q>} \{<Rd>,\} <Rn>, \#<const>

**ANDS (S == 1)**

ANDS\{<c>\}{<q>} \{<Rd>,\} <Rn>, \#<const>

\[d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1');\]
\[(\text{imm32}, \text{carry}) = \text{A32ExpandImm}_C(\text{imm12}, \text{PSTATE}.C);\]

T1

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 1 & 0 & i & 0 & 0 & 0 & 0 & S & Rn & 0 & \text{imm3} & Rd & \text{imm8} \\
\end{array}
\]

**AND (S == 0)**

AND\{<c>\}{<q>} \{<Rd>,\} <Rn>, \#<const>

**ANDS (S == 1 &\& Rd != 1111)**

ANDS\{<c>\}{<q>} \{<Rd>,\} <Rn>, \#<const>

\[\text{if Rd == '1111' &\& S == '1' then SEE "TST (immediate)";}\]
\[d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1');\]
\[(\text{imm32}, \text{carry}) = \text{T32ExpandImm}_C(i:imm3:imm8, \text{PSTATE}.C);\]
\[\text{if (d == 15 &\& \!setflags) \| n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}\]

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c>
See Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<Rd>
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn>
For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const>
For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AND, ANDS (register)

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

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

\[
\begin{array}{cccccccccccc}
! = 1111 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S & Rn & Rd & imm5 & stype & 0 & Rm \\
\end{array}
\]

AND, rotate right with extend (S == 0 & imm5 == 0000 & stype == 11)

\[
\text{AND}\{<c>\}\{<q>\} \{<Rd>, \} \{<Rn>, \} \{<Rm>, \} \{RRX}
\]

AND, shift or rotate by value (S == 0 & !(imm5 == 0000 & stype == 11))

\[
\text{AND}\{<c>\}\{<q>\} \{<Rd>, \} \{<Rn>, \} \{<Rm>, \} \{<shift> \#<amount>\}
\]

ANDS, rotate right with extend (S == 1 & imm5 == 0000 & stype == 11)

\[
\text{ANDS}\{<c>\}\{<q>\} \{<Rd>, \} \{<Rn>, \} \{<Rm>, \} \{RRX}
\]

ANDS, shift or rotate by value (S == 1 & !(imm5 == 0000 & stype == 11))

\[
\text{ANDS}\{<c>\}\{<q>\} \{<Rd>, \} \{<Rn>, \} \{<Rm>, \} \{<shift> \#<amount>\}
\]

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ \text{setflags} = (S == '1'); \ (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5});
\]

T1

\[
\begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

T1

\[
\text{AND}\{<c>\}\{<q>\} \{<Rdn>, \} \{<Rm>, \} \{\text{Inside IT block}\}
\]

\[
\text{ANDS}\{<c>\}\{<q>\} \{<Rdn>, \} \{<Rm>, \} \{\text{Outside IT block}\}
\]

\[
d = \text{UInt}(Rdn); \ n = \text{UInt}(Rdn); \ m = \text{UInt}(Rm); \ \text{setflags} = \text{InITBlock}(); \ (\text{shift}_t, \text{shift}_n) = (\text{SRTypenLSL}, \ 0);
\]

T2
AND, rotate right with extend (S == 0 & imm3 == 000 & imm2 == 00 & stype == 11)

\[
\text{AND}\{<c>\}\{<q>\} \{<Rd>, \}<Rn>, <Rm>, \text{RRX}
\]

AND, shift or rotate by value (S == 0 & !(imm3 == 000 & imm2 == 00 & stype == 11))

\[
\text{AND}\{<c>\}.W \{<Rd>, \}<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
\]

\[
\text{AND}\{<c>\}\{<q>\} \{<Rd>, \}<Rn>, <Rm> \{, <\text{shift}> #<\text{amount}>\}
\]

ANDS, rotate right with extend (S == 1 & imm3 == 000 & Rd != 1111 & imm2 == 00 & stype == 11)

\[
\text{ANDS}\{<c>\}\{<q>\} \{<Rd>, \}<Rn>, <Rm>, \text{RRX}
\]

ANDS, shift or rotate by value (S == 1 & !(imm3 == 000 & imm2 == 00 & stype == 11) & Rd != 1111)

\[
\text{ANDS.W}\{<Rd>, \}<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
\]

\[
\text{ANDS}\{<c>\}\{<q>\} \{<Rd>, \}<Rn>, <Rm> \{, <\text{shift}> #<\text{amount}>\}
\]

\[
\text{if Rd == '1111' & S == '1' then SEE "TST (register)";}
\]

\[
d = \text{UInt}(Rd);\ n = \text{UInt}(Rn);\ m = \text{UInt}(Rm);\ \text{setflags} = (S == '1');
\]

\[
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm3:imm2});
\]

\[
\text{if (d == 15 & !setflags) || n == 15 || m == 15 then UNPREDICTABLE;}
\]

// Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<\text{c}> See Standard assembler syntax fields.

<\text{q}> See Standard assembler syntax fields.

<\text{Rdn}> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<\text{Rd}> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

\begin{itemize}
  \item For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  \item For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
\end{itemize}

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<\text{Rn}> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<\text{Rm}> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<\text{shift}> Is the type of shift to be applied to the second source register, encoded in "stype":

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

AND, ANDS (register)
<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if ANDS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ANDS <Rd>, <Rn> had been written.
- Inside an IT block, if AND<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though AND<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    if d == 15 then        // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

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

### Flag setting (S == 1)

ANDS\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>, <shift> <Rs>

### Not flag setting (S == 0)

ANDS\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>, <shift> <Rs>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler Symbols

| <c>   | See Standard assembler syntax fields. |
| <q>   | See Standard assembler syntax fields. |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn>  | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm>  | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in "stype": |
| stype | <shift> |
| 00    | LSL    |
| 01    | LSR    |
| 10    | ASR    |
| 11    | ROR    |
| <Rs>  | Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);  
        PSTATE.C = carry;
        // PSTATE.V unchanged
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

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

This is an alias of MOV, MOVs (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVs (register).
- The description of MOV, MOVs (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

```
| ! = 1111 | 0 0 0 1 1 0 | Rd |
| cond | S | imm5 | 1 0 | Rm |
```

MOV, shift or rotate by value

ASR{<c>}<{<q>} {<Rd>,} <Rm>, #<imm>

is equivalent to

MOV{<c>}<{<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

T2

```
| 0 0 0 1 0 | imm5 | Rd |
| op |
```

T2

ASR<c>{<q>} {<Rd>,} <Rm>, #<imm> // (Inside IT block)

is equivalent to

MOV<c>{<q>} <Rd>, <Rm>, ASR #<imm>

and is the preferred disassembly when InITBlock().

T3

```
| 1 1 1 0 1 1 1 0 0 1 | Rd |
| S | imm3 | 1 0 | Rm |
```

MOV, shift or rotate by value

ASR<c>.W <{<Rd>,} <Rm>, #<imm> // (Inside IT block, and <Rd>, <Rm>, #<imm> can be represented in T2)

ASR{<c>}<{<q>} <Rd>,} <Rm>, #<imm>

is equivalent to

MOV{<c>}<{<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.
Assembler Symbols

See Standard assembler syntax fields.

For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
**ASR (register)**

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of **MOV, MOVs (register-shifted register)**. This means:

- The encodings in this description are named to match the encodings of **MOV, MOVs (register-shifted register)**.
- The description of **MOV, MOVs (register-shifted register)** gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A₁) and T32 (T₁ and T₂).

### A₁

| ! = 1111 | 0 0 0 1 1 0 1 0 | (0) (0) (0) | Rd | Rs | 0 1 0 1 | Rm |
| cond | S | stype |

**Not flag setting**

\[
\text{ASR} \{<c>\}<q> \{<Rd>,} \ <Rm>, \ <Rs> \\
\text{is equivalent to} \\
\text{MOV} \{<c>\}<q> \ <Rd>,} \ <Rm>, \ ASR \ <Rs> \\
\text{and is always the preferred disassembly.}
\]

### T₁

| 0 1 0 0 0 0 0 1 0 0 | Rs | Rdm |
| op |

**Arithmetic shift right**

\[
\text{ASR}<c>\{<q>\} \{<Rdm>,} \ <Rdm>,} \ <Rs> \ // \text{ (Inside IT block)} \\
\text{is equivalent to} \\
\text{MOV}<c>\{<q>\} \ <Rdm>,} \ <Rdm>,} \ ASR \ <Rs> \\
\text{and is the preferred disassembly when InITBlock().}
\]

### T₂

| 1 1 1 1 1 0 1 0 0 | Rm |
| 1 1 1 1 | Rd |
| 0 0 0 0 | Rs |

**Not flag setting**

\[
\text{ASR}<c>.W \ {<Rd>,} \ <Rm}, \ <Rs> \ // \text{ (Inside IT block, and }<Rd>,} \ <Rm>,} \ <shift>,} \ <Rs> \text{ can be represented in T₁)} \\
\text{ASR}<c>\{<q>\} \ {<Rd>,} \ <Rm}, \ <Rs> \\
\text{is equivalent to}
\]
MOV \{<c>\}{<q>\} <Rd>, <Rm>, ASR <Rs>

and is always the preferred disassembly.

**Assembler Symbols**

\(<c>\) See *Standard assembler syntax fields*.

\(<q>\) See *Standard assembler syntax fields*.

\(<Rdm>\) Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rm>\) Is the first general-purpose source register, encoded in the "Rm" field.

\(<Rs>\) Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation**

The description of **MOV, MOVS (register-shifted register)** gives the operational pseudocode for this instruction.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ASRS (immediate)

Arithmetic Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register. If the destination register is not the PC, this instruction updates the condition flags based on the result. The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | (0)(0)(0)(0) | Rd | imm5 | 1 | 0 | 0 | Rm |
| cond | S | stype |

MOVS, shift or rotate by value

ASRS{<c>}{<q>} {<Rd>}, <Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | |
| op |

T2

ASRS{<q>} {<Rd>}, <Rm>, #<imm> // (Outside IT block)

is equivalent to

MOV{<q>} <Rd>, <Rm>, ASR #<imm>

and is the preferred disassembly when !InITBlock().

T3

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | (0) | imm3 | Rd | imm2 | 1 | 0 | Rm |
| S | stype |
MOVS, shift or rotate by value

ASRS.W {<Rd>,} <Rm>, #<imm> // (Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)

ASRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

<imm> For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
ASRS (register)

Arithmetic Shift Right, setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

```
<cond> Rd Rs Rm <c> <q> S stype
```

Flag setting

\[
\text{ASRS}\{<c>\}{<q>}\{<Rd>,,<Rm>,,<Rs>}
\]

is equivalent to

\[
\text{MOV}\{<c>\}{<q>}\{<Rd>,,<Rm>,,\text{ASR}<Rs>}
\]

and is always the preferred disassembly.

T1

```
<op> Rs Rdm int Rd Rm Rs
```

Arithmetic shift right

\[
\text{ASRS}\{<q>\}\{<Rdm>,,<Rdm>,,<Rs>\} // (Outside IT block)
\]

is equivalent to

\[
\text{MOV}\{<q>\}\{<Rdm>,,<Rdm>,,\text{ASR}<Rs>\}
\]

and is the preferred disassembly when !InITBlock().

T2

```
<int> Rd Rm Rs
```

Flag setting

\[
\text{ASRS.W}\{<Rd>,,<Rm>,,<Rs>\} // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
\]

\[
\text{ASRS}\{<c>\}{<q>}\{<Rd>,,<Rm>,,<Rs>}
\]

is equivalent to
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>

and is always the preferred disassembly.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rdm> Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> Is the first general-purpose source register, encoded in the "Rm" field.

<Rs> Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
B

Branch causes a branch to a target address.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3 and T4).

A1

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

\[ \text{cond} \]

\[ \text{imm24} \]

A1

\[ \text{B\{<c>\}{<q> \{<label>\}}} \]

\[ \text{imm32} = \text{SignExtend}(\text{imm24:'00'}, 32); \]

T1

\[ \begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array} \]

\[ \text{cond} \]

\[ \text{imm8} \]

T1

\[ \text{B\{<c>\}{<q> \{<label>\}}} // (Not permitted in IT block) \]

\[ \text{if cond == '1110' then SEE "UDF";} \]

\[ \text{if cond == '1111' then SEE "SVC";} \]

\[ \text{imm32} = \text{SignExtend}(\text{imm8:'0'}, 32); \]

\[ \text{if InITBlock()} \text{ then UNPREDICTABLE;} \]

T2

\[ \begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array} \]

\[ \text{imm11} \]

T2

\[ \text{B\{<c>\}{<q> \{<label>\}}} // (Outside or last in IT block) \]

\[ \text{imm32} = \text{SignExtend}(\text{imm11:'0'}, 32); \]

\[ \text{if InITBlock()} \&\& !\text{LastInITBlock()} \text{ then UNPREDICTABLE;} \]

T3

\[ \begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array} \]

\[ \text{cond} \]

\[ \text{imm6} \]

\[ \text{imm11} \]

T3

\[ \text{B\{<c>\}.W \{<label>\}}} // (Not permitted in IT block, and <label> can be represented in T1) \]

\[ \text{B\{<c>\}{<q> \{<label>\}}} // (Not permitted in IT block) \]

\[ \text{if cond<3:1> == '111' then SEE "Related encodings";} \]

\[ \text{imm32} = \text{SignExtend}(\text{S:J2:J1:imm6:imm11:'0'}, 32); \]

\[ \text{if InITBlock()} \text{ then UNPREDICTABLE;} \]
B{<c>}.W <label> // (<label> can be represented in T2)
B{<c>}{<q>} <label>
I1 = NOT(J1 XOR S);  I2 = NOT(J2 XOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.
Related encodings: Branches and miscellaneous control.

Assembler Symbols

<c> For encoding A1, T2 and T4: see Standard assembler syntax fields.
For encoding T1: see Standard assembler syntax fields. Must not be AL or omitted.
For encoding T3: see Standard assembler syntax fields. <c> must not be AL or omitted.

<q> See Standard assembler syntax fields.

<label> For encoding A1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are multiples of 4 in the range –33554432 to 33554428.
For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –256 to 254.
For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –2048 to 2046.
For encoding T3: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –1048576 to 1048574.
For encoding T4: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32, BranchType_DIR);
**BFC**

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>1111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>msb</td>
<td>Rd</td>
<td>lsb</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cond

**A1**

\[
\text{BFC}\{\text{<c>}}\{\text{<q>}}\{\text{<Rd>}}\{\#\text{<lsb>}}\{\#\text{<width>}}
\]

\[
d = \text{UInt}(\text{Rd}); \msbit = \text{UInt}(\text{msb}); \lsbit = \text{UInt}(\text{lsb});
\]

if \( d == 15 \) then UNPREDICTABLE;

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(0)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>imm3</td>
<td>Rd</td>
<td>imm2</td>
</tr>
</tbody>
</table>

**T1**

\[
\text{BFC}\{\text{<c>}}\{\text{<q>}}\{\text{<Rd>}}\{\#\text{<lsb>}}\{\#\text{<width>}}
\]

\[
d = \text{UInt}(\text{Rd}); \msbit = \text{UInt}(\text{msb}); \lsbit = \text{UInt}(\text{imm3:imm2});
\]

if \( d == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see **Architectural Constraints on UNPREDICTABLE behaviors**.

**Assembler Symbols**

- \(<\text{c}>\) See *Standard assembler syntax fields*.
- \(<\text{q}>\) See *Standard assembler syntax fields*.
- \(<\text{Rd}>\) Is the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{lsb}>\) For encoding A1: is the least significant bit to be cleared, in the range 0 to 31, encoded in the "lsb" field.
  
  For encoding T1: is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the "imm3:imm2" field.
- \(<\text{width}>\) Is the number of bits to be cleared, in the range 1 to 32.<lsb>, encoded in the "msb" field as <lsb>+<width>-1.  

**Operation**

if \( \text{ConditionPassed()} \) then

\[
\text{EncodingSpecificOperations();}
\]

if \( \text{msbit} >= \text{lsbit} \) then

\[
\text{R}[d]\langle\text{msbit:lsbit}\rangle = \text{Replicate}(\text{'0'}, \text{msbit}-\text{lsbit}+1);
\]

// Other bits of R[d] are unchanged

else

\[
\text{UNPREDICTABLE;}
\]
**CONstrained Unpredictable Behavior**

If \( \text{msbit} < \text{lsbit} \), then one of the following behaviors must occur:

- The instruction is **undefined**.
- The instruction executes as **NOP**.
- The value in the destination register is **unknown**.

**Operational Information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

![Register Map]

### A1

$$\text{BFI}\{<c>\}\{<q>\} \ <Rd>, \ <Rn>, \ #<\text{lsb}>, \ #<\text{width}>$$

if \(Rn == \text{'b1111'}\) then SEE “BFC”;

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{msbit} = \text{UInt}(\text{msb}); \quad \text{lsbit} = \text{UInt}(\text{lsb});
\]

if \(d == 15\) then UNPREDICTABLE;

### T1

$$\text{BFI}\{<c>\}\{<q>\} \ <Rd>, \ <Rn>, \ #<\text{lsb}>, \ #<\text{width}>$$

if \(Rn == \text{'b1111'}\) then SEE “BFC”;

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{msbit} = \text{UInt}(\text{msb}); \quad \text{lsbit} = \text{UInt}(\text{imm3:imm2});
\]

if \(d == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>`
  
  See [Standard assembler syntax fields](#).

- `<q>`
  
  See [Standard assembler syntax fields](#).

- `<Rd>`
  
  Is the general-purpose destination register, encoded in the "Rd" field.

- `<Rn>`
  
  Is the general-purpose source register, encoded in the "Rn" field.

- `<lsb>`
  
  For encoding A1: is the least significant destination bit, in the range 0 to 31, encoded in the "lsb" field.

  For encoding T1: is the least significant destination bit, in the range 0 to 31, encoded in the "imm3:imm2" field.

- `<width>`
  
  Is the number of bits to be copied, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1.
Operation

if \textbf{ConditionPassed}() then
    EncodingSpecificOperations();
    if msbit \geq lsbit then
        R[d]<msbit:lsbit> = R[n]<\text{msbit-\text{lsbit}}:0>;
        // Other bits of R[d] are unchanged
    else
        \textbf{UNPREDICTABLE};

CONSTRAINED UNPREDICTABLE behavior

If msbit < lsbit, then one of the following behaviors must occur:

- The instruction is \textbf{UNDEFINED}.
- The instruction executes as \textbf{NOP}.
- The value in the destination register is \textbf{UNKNOWN}.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

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

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|----------------|
| != 1111 | 0 0 1 1 1 1 0 | S | Rn | Rd | imm12 |
```

cond

**BIC (S == 0)**

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**BICS (S == 1)**

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

```plaintext
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|---------------|---------------|
| 1 | 1 | 1 | 1 | 0 | i | 0 | 0 | 0 | 0 | 1 | S | Rn | 0 | imm3 | Rd | imm8 |
```

**BIC (S == 0)**

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**BICS (S == 1)**

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

```plaintext
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
See Standard assembler syntax fields.

For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to a address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND NOT(imm32);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Bitwise Bit Clear (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.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores `PSTATE` from `SPSR_<current_mode>`.
  - The PE checks `SPSR_<current_mode>` for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is *UNDEFINED* in Hyp mode.
  - The instruction is *CONSTRAINED* *UNPREDICTABLE* in User mode and System mode.

It has encodings from the following instruction sets: A32 (*A1*) and T32 (*T1* and *T2*).

### A1

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0  | 0  | 0  | 1  | 1  | 0  | S  | Rn | Rd | imm5 | stype | 0  | Rm |

cond
```

**BIC, rotate right with extend (S == 0 & imm5 == 00000 & stype == 11)**

BIC{<c}>{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**BIC, shift or rotate by value (S == 0 & !(imm5 == 00000 & stype == 11))**

BIC{<c}>{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

**BICS, rotate right with extend (S == 1 & imm5 == 00000 & stype == 11)**

BICS{<c}>{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**BICS, shift or rotate by value (S == 1 & !(imm5 == 00000 & stype == 11))**

BICS{<c}>{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

```
d = Uint(Rd); n = Uint(Rn); m = Uint(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

### T1

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>Rdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**T1**

BIC{<c>} {<Rdn>,} <Rdn>, <Rm> // (Inside IT block)

BICS{<c>} {<Rdn>,} <Rdn>, <Rm> // (Outside IT block)

```
d = Uint(Rdn); n = Uint(Rdn); m = Uint(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

**T2**
BIC, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

BIC{c}{q} {<Rd>,} <Rn>, <Rm>, RRX

BIC, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

BIC<c>.W {<Rd>,} <Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
BIC{c}{q} {<Rd>,} <Rn>, <Rm} {, <shift> #=><amount>}

BICS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

BICS{c}{q} {<Rd>,} <Rn>, <Rm>, RRX

BICS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

BICS.W {<Rd>,} <Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
BICS{c}{q} {<Rd>,} <Rn>, <Rm} {, <shift> #=><amount>}

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rdn>` Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  - For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
  - For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.
- `<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
  - For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
  - For encoding T2: is the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the type of shift to be applied to the second source register, encoded in "stype":

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

- `<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when `<shift>` = LSL or ROR) or 1 to 32 (when `<shift>` = LSR or ASR), encoded in the "imm5" field as `<amount>` modulo 32.
For encoding T2: is the shift amount, in the range 1 to 31 (when \(<\text{shift}> = \text{LSL or ROR}\) or 1 to 32 (when \(<\text{shift}> = \text{LSR or ASR}\)\), encoded in the “imm3:imm2” field as \(<\text{amount}> \mod 32\).

**Operation**

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations();
  (shifted, carry) = \text{Shift}_C(R[m], \text{shift}_t, \text{shift}_n, \text{PSTATE}.C);
  result = R[n] AND NOT(shifted);
  if d == 15 then // Can only occur for A32 encoding
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
  else
    R[d] = result;
    if setflags then
      PSTATE.N = result<31>;
      PSTATE.Z = IsZeroBit(result);
      PSTATE.C = carry;
      // PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

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

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

A1

| !=1111 | 0 0 0 1 1 1 0 S | Rn | Rd | Rs | 0 | stype | 1 | Rm |
|cond|

Flag setting (S == 1)

BICS{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>, <shift> <Rs>}

Not flag setting (S == 0)

BIC{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>, <shift> <Rs>}

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

见 Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in "stype":

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

<Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Breakpoint causes a Breakpoint Instruction exception. Breakpoint is always unconditional, even when inside an IT block.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

$$\begin{array}{cccccccccccccccc}
\hline
\text{cond} &= 1111 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & \text{imm12} & 0 & 1 & 1 & 1 & \text{imm4} & \\
\end{array}$$

$$\text{if cond} \neq '1110' \text{ then UNPREDICTABLE; // BKPT must be encoded with AL condition}$$

T1

$$\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\text{imm8} &= 0 & 1 & 1 & 1 & 1 & 0 \\
\end{array}$$

$$\text{imm16} = \text{ZeroExtend}(\text{imm8}, 16);$$

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q> See Standard assembler syntax fields. An BKPT instruction must be unconditional.

<imm> For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Is ignored otherwise.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Is ignored otherwise.
Operation

EncodingSpecificOperations();
AArch32.SoftwareBreakpoint(imm16);
BL, BLX (immediate)

Branch with Link calls a subroutine at a PC-relative address, and setting LR to the return address. Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, setting LR to the return address, and changes the instruction set from A32 to T32, or from T32 to A32.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
!= 1111 | 1 0 1 1

cond
```

```
A1
BL{<c>{q}}<label>

imm32 = SignExtend(imm24:'00', 32); targetInstrSet = InstrSet_A32;
```

A2

```
31 30 29 28 27 26 25 24 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 1 H

cond
```

```
A2
BLX{<c>{q}}<label>

imm32 = SignExtend(imm24:H:'0', 32); targetInstrSet = InstrSet_T32;
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 S

imm10 | 1 1 | j1 | j2 |

imm11
```

```
T1
BL{<c>{q}}<label>

I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32); targetInstrSet = InstrSet_T32;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T2

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 S

imm10H | 1 1 | j1 | j2 |

imm10L H
```

```
T2
BLX{<c>{q}}<label>

if H == '1' then UNDEFINED;
I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10H:imm10L:'00', 32);
targetInstrSet = InstrSet_A32;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```
For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` For encoding A1, T1 and T2: see *Standard assembler syntax fields*. For encoding A2: see *Standard assembler syntax fields*. `<c>` must be AL or omitted.

- `<q>` See *Standard assembler syntax fields*.

- `<label>` For encoding A1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are multiples of 4 in the range –33554432 to 33554428.

- For encoding A2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –33554432 to 33554430.

- For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214.

- For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the Align(PC, 4) value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are multiples of 4 in the range –16777216 to 16777212.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_A32 then
        LR = PC - 4;
    else
        LR = PC<31:1> : '1';
    if targetInstrSet == InstrSet_A32 then
        targetAddress = Align(PC,4) + imm32;
    else
        targetAddress = PC + imm32;
    SelectInstrSet(targetInstrSet);
    BranchWritePC(targetAddress, BranchType_DIRCALL);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address specified in the register, and if necessary changes to the instruction set indicated by bit[0] of the register value. If the value in bit[0] is 0, the instruction set after the branch will be A32. If the value in bit[0] is 1, the instruction set after the branch will be T32.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | [(1)] | [(1)] | [(1)] | [(1)] | [(1)] | [(1)] | [(1)] | [(1)] | [(1)] | [(1)] | 0 | 0 | 0 | 1 | Rm |

cond

A1

BLX{<c>}{<q>} <Rm>

m = UInt(Rm);
if m == 15 then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Rm | (0) | (0) | (0) |

T1

BLX{<c>}{<q>} <Rm>

m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InitITBlock() && !LastInITBlock() then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.
<Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

Operation

if ConditionPassed() then
   EncodingSpecificOperations();
   target = R[m];
   if CurrentInstrSet() == InstrSet_A32 then
      next_instr_addr = PC - 4;
      LR = next_instr_addr;
   else
      next_instr_addr = PC - 2;
      LR = next_instr_addr<31:1> : '1';
   BXWritePC(target, BranchType_INDCALL);
Branch and Exchange causes a branch to an address and instruction set specified by a register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

\[
\text{BX}\{<c>\}{<q>} <Rm>
\]

\[
m = \text{UInt}(Rm);
\]

T1

\[
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & Rm & |(0)|(0)|(0)
\]

T1

\[
\text{BX}\{<c>\}{<q>} <Rm>
\]

\[
m = \text{UInt}(Rm);
\]

\[
\text{if } \text{InITBlock}() \&\& !\text{LastInITBlock}() \text{ then UNPREDICTABLE;}
\]

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\[
<\text{c}> \quad \text{See Standard assembler syntax fields.}
\]

\[
<\text{q}> \quad \text{See Standard assembler syntax fields.}
\]

\[
<\text{Rm}> \quad \text{For encoding A1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used.}
\]

\[
\text{For encoding T1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used.}
\]

\[
\text{If <Rm> is the PC at a non word-aligned address, it results in UNPREDICTABLE behavior because the address passed to the BXWritePC() pseudocode function has bits}<1:0> = '10'.
\]

Operation

\[
\text{if } \text{ConditionPassed}() \text{ then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{BXWritePC(R[m], BranchType_INDIR)};
\]
BXJ

Branch and Exchange, previously Branch and Exchange Jazelle.
In Armv8, BXJ behaves as a BX instruction, see BX. This means it causes a branch to an address and instruction set specified by a register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Rm |
| cond |

A1

BXJ{<c>}{<q>} <Rm>

m = UInt(Rm);
if m == 15 then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

T1

BXJ{<c>}{<q>} <Rm>

m = UInt(Rm);
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<e> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.
<Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m], BranchType_INDIR);

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CBNZ, CBZ

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>op</td>
<td>0</td>
<td>i</td>
<td>1</td>
<td>imm5</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CBNZ (op == 1)

```
CBNZ{<q>} <Rn>, <label>
```

CBZ (op == 0)

```
CBZ{<q>} <Rn>, <label>
```

\[ n = \text{UInt}(Rn); \text{imm32} = \text{ZeroExtend}(i:imm5:'0', 32); \text{nonzero} = (\text{op} == '1'); \]

if \( \text{InITBlock}() \) then \text{UNPREDICTABLE};

For more information about the \text{CONSTRAINED UNPREDICTABLE} behavior, see \text{Architectural Constraints on UNPREDICTABLE behaviors}.

Assembler Symbols

- `<q>`: See \textit{Standard assembler syntax fields}.
- `<Rn>`: Is the general-purpose register to be tested, encoded in the "Rn" field.
- `<label>`: Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 and in the range 0 to 126, is encoded as "i:imm5" times 2.

Operation

```
Encodingspecificoperations();
if nonzero != IsZero(R[n]) then
    Branchwritepc(PC + imm32, BranchType_DIR);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CLREX

Clear-Exclusive clears the local monitor of the executing PE.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 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 1 0 1 0 1 1 1 (1)(1)(1)(1)(1)(1)(1)(1)(1)(0)(0)(0)(0) 0 0 0 1 (1)(1)(1)(1)
```

CLREX{<c>}{<q>}

// No additional decoding required

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 1 1 0 1 1 (1)(1)(1)(1)(1) 1 0 (0) 0 (1)(1)(1)(1)(1) 0 0 1 0 (1)(1)(1)
```

CLREX{<c>}{<q>}

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. Must be AL or omitted.
    For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**CLZ**

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

| != 1111 | 0 0 0 1 0 1 1 0 | (1)(1)(1)(1) | Rd | (1)(1)(1)(1) | 0 0 0 1 | Rm |

cond

### A1

`CLZ{<c>}{<q>}{<Rd>},{<Rm>}`

```plaintext
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

### T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 1 1 1 1 1 1 0 1 0 1 0 1 1 | Rd | (1)(1)(1)(1) | 0 1 0 0 | Rm |

### T1

`CLZ{<c>}{<q>}{<Rd>},{<Rm>}`

```plaintext
d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

**CONstrained UNPREDICTABLE behavior**

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes with the additional decode: `m = UInt(Rn)`.
- The value in the destination register is UNKNOWN.

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field.
  For encoding T1: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

**Operation**

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations();
  result = CountLeadingZeroBits(R[m]);
  R[d] = result<31:0>;
```
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccccccccccccccc}
\hline
!= 1111 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & \text{Rn} & \{0\} & \{0\} & \{0\} & \text{imm12} & \text{cond} \\
\end{array}
\]

A1

CMN{<c>}{<q>} <Rn>, #<const>

\[n = \text{UInt}(\text{Rn}); \quad \text{imm32} = \text{A32ExpandImm}(\text{imm12});\]

T1

\[
\begin{array}{cccccccccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 1 & 0 & i & 0 & 1 & 0 & 0 & 0 & 1 & \text{Rn} & 0 & \text{imm3} & 1 & 1 & 1 & 1 & \text{imm8} \\
\end{array}
\]

T1

CMN{<c>}{<q>} <Rn>, #<const>

\[n = \text{UInt}(\text{Rn}); \quad \text{imm32} = \text{T32ExpandImm}(i:imm3:imm8);\]

if \(n == 15\) then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- <c> See Standard assembler syntax fields.
- <q> See Standard assembler syntax fields.
- <Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
  For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
- <const> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.
  For encoding T1: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  \((\text{result}, \text{nzcv}) = \text{AddWithCarry}(R[n], \text{imm32}, ‘0’);\)
  \(\text{PSTATE.<N,Z,C,V>} = \text{nzcv};\)

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>[0]</td>
<td>[0]</td>
<td>[0]</td>
<td>[0]</td>
<td>imm5</td>
<td>stype</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cond

Rotate right with extend (imm5 == 00000 && stype == 11)

CMN{<c>}{<q>} <Rn>, <Rm>, RRX

Shift or rotate by value (! (imm5 == 00000 && stype == 11))

CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

n = UInt(Rn); m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | Rm | Rn |

T1

CMN{<c>}{<q>} <Rn>, <Rm>

n = UInt(Rn); m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);

T2

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | Rn | [0] | imm3 | 1 | 1 | 1 | 1 | imm2 | stype | Rm |

Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

CMN{<c>}{<q>} <Rn>, <Rm>, RRX

Shift or rotate by value (! (imm3 == 000 && imm2 == 00 && stype == 11))

CMN{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)

CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

n = UInt(Rn); m = UInt(Rm);
(shift t, shift n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

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

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

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

Compare Negative (register-shifted register) adds a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 & 0 0 0 1 0 1 1 1 | Rn | [0][0][0][0] | Rs | 0 | stype | 1 | Rm |

cond

A1

CMN{<c>}{<q>}{<Rn>}{<Rm>}{<type>}{<Rs>}

n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<type>` Is the type of shift to be applied to the second source register, encoded in “stype”:

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

- `<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, ‘0’);
    PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

**A1**

<table>
<thead>
<tr>
<th>cond</th>
<th>Rn</th>
<th>imm12</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>0</td>
<td>0 1 1 0 1 0 1</td>
</tr>
</tbody>
</table>

\[n = \text{UInt}(\text{Rn}); \text{ imm32 } = \text{A32ExpandImm}(\text{imm12});\]

**T1**

<table>
<thead>
<tr>
<th>imm8</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

\[n = \text{UInt}(\text{Rn}); \text{ imm32 } = \text{ZeroExtend}(\text{imm8}, 32);\]

**T2**

<table>
<thead>
<tr>
<th>imm8</th>
<th>imm3</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>i 0 1 1 0 1 1</td>
<td>1 1 1 1</td>
<td>imm8</td>
</tr>
</tbody>
</table>

\[n = \text{UInt}(\text{Rn}); \text{ imm32 } = \text{T32ExpandImm}(\text{i:imm3:imm8});\]

if \(n == 15\) then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
- For encoding T1: is a general-purpose source register, encoded in the "Rn" field.
- For encoding T2: is the general-purpose source register, encoded in the "Rn" field.
- `<imm8>` Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
- `<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.
For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

A1

\[
\begin{array}{cccccccccccccc}
\hline
Rn & [0] & [0] & [0] & [0] & imm5 & stype & 0 & Rm \\
\end{array}
\]

cond

Rotate right with extend (imm5 == 00000 && stype == 11)

CMP{<c>}<q> <Rn>, <Rm>, RRX

Shift or rotate by value (!(imm5 == 00000 && stype == 11))

CMP{<c>}<q> <Rn>, <Rm> {, <shift> #<amount>}

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

\[
(shift_t, shift_n) = \text{DecodeImmShift}(stype, imm5);
\]

T1

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
Rm & Rn \\
\end{array}
\]

T1

CMP{<c>}<q> <Rn>, <Rm> // (<Rn> and <Rm> both from R0-R7)

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

\[
(shift_t, shift_n) = (\text{SRType_LSL}, 0);
\]

T2

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
N & Rm & Rn \\
\end{array}
\]

T2

CMP{<c>}<q> <Rn>, <Rm> // (<Rn> and <Rm> not both from R0-R7)

\[
n = \text{UInt}(N:Rn); \ m = \text{UInt}(Rm);
\]

\[
(shift_t, shift_n) = (\text{SRType_LSL}, 0);
\]

if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If n < 8 && m < 8, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The condition flags become UNKNOWN.

T3
Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

CMP{<c>}{<q>} <Rn>, <Rm>, RRX

Shift or rotate by value (!imm3 == 000 && imm2 == 00 && stype == 11))

CMP{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1 or T2)

n = UInt(Rn);  m = UInt(Rm);
(amount, shift) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<...> See Standard assembler syntax fields.
<...> See Standard assembler syntax fields.
<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For encoding T1 and T3: is the first general-purpose source register, encoded in the "Rn" field.
For encoding T2: is the first general-purpose source register, encoded in the "N:Rn" field.
<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For encoding T1, T2 and T3: is the second general-purpose source register, encoded in the "Rm" field.
<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

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

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Compare (register-shifted register) subtracts a register-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

A1

\[
\begin{array}{ccccccccccccccccccccccc}
\hline
!= & 1111 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & Rn & [0][0][0][0] & Rs & 0 & stype & 1 & Rm & cond
\end{array}
\]

A1

\[
\text{CMP}\{<c>\}{<q>}, <Rm>, <type> <Rs>
\]

\[
n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ s = \text{UInt}(Rs);
\text{shift}_t = \text{DecodeRegShift}(stype);
\text{if } n == 15 || m == 15 || s == 15 \text{ then UNPREDICTABLE;}
\]

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

\(<type>\) Is the type of shift to be applied to the second source register, encoded in "stype":

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

\(<Rs>\) Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
   EncodingSpecificOperations();
   shift_n = UInt(R[s]<7:0>);
   shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
   (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
   PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Change PE State changes one or more of the \textit{PSTATE}.{A, I, F} interrupt mask bits and, optionally, the \textit{PSTATE}.M mode field, without changing any other \textit{PSTATE} bits.

CPS is treated as \texttt{NOP} if executed in User mode unless it is defined as being \texttt{CONSTRAINED UNPREDICTABLE} elsewhere in this section.

The PE checks whether the value being written to \textit{PSTATE}.M is legal. See \textit{Illegal changes to PSTATE.M}.

It has encodings from the following instruction sets: A32 (\texttt{A1}) and T32 (\texttt{T1} and \texttt{T2}).

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

Change mode (imod == 00 && M == 1)

\begin{verbatim}
CPS{<q>} #<mode> // (Cannot be conditional)
\end{verbatim}

Interrupt disable (imod == 11 && M == 0)

\begin{verbatim}
CPSID{<q>} <iflags> // (Cannot be conditional)
\end{verbatim}

Interrupt disable and change mode (imod == 11 && M == 1)

\begin{verbatim}
CPSID{<q>} <iflags>, #<mode> // (Cannot be conditional)
\end{verbatim}

Interrupt enable (imod == 10 && M == 0)

\begin{verbatim}
CPSIE{<q>} <iflags> // (Cannot be conditional)
\end{verbatim}

Interrupt enable and change mode (imod == 10 && M == 1)

\begin{verbatim}
CPSIE{<q>} <iflags>, #<mode> // (Cannot be conditional)
\end{verbatim}

\begin{verbatim}
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UNPREDICTABLE;
\end{verbatim}

CONSTRAINED UNPREDICTABLE behavior

If imod == '01', then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as \texttt{NOP}.

If imod == '00' && M == '0', then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as \texttt{NOP}.

If mode != '00000' && M == '0', then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as \texttt{NOP}.
- The instruction executes with the additional decode: changemode = TRUE.
- The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.
If \texttt{imod<1> == '1' \&\& A:I:F == '000'}, then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as \texttt{NOP}.
- The instruction behaves as if \texttt{imod<1> == '0'}.
- The instruction behaves as if A:I:F has an \texttt{UNKNOWN} nonzero value.

If \texttt{imod<1> == '0' \&\& A:I:F != '000'}, then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as \texttt{NOP}.
- The instruction behaves as if \texttt{imod<1> == '1'}.
- The instruction behaves as if A:I:F == '000'.

\begin{center}
\textbf{T1}
\end{center}

\begin{verbatim}
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 1 0 1 1 0 0 1 1 |im(0)| A | I | F
\end{verbatim}

\textbf{Interrupt disable (im == 1)}

\texttt{CPSID{<q>}} \texttt{<iflags>} // (Not permitted in IT block)

\textbf{Interrupt enable (im == 0)}

\texttt{CPSIE{<q>}} \texttt{<iflags>} // (Not permitted in IT block)

\begin{verbatim}
if A:I:F == '000' then UNPREDICTABLE;
enable = (im == '0'); disable = (im == '1'); changemode = FALSE;
affectA = (A == '1'); affectI = (I == '1'); affectF = (F == '1');
if \texttt{InITBlock()} then UNPREDICTABLE;
\end{verbatim}

\textbf{CONSTRAINED UNPREDICTABLE behavior}

If A:I:F == '000', then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as \texttt{NOP}.

\begin{center}
\textbf{T2}
\end{center}

\begin{verbatim}
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 1 0 1 0 |(1)(1)(1)(1)| 1 0 |0|0|0|0|mod |M|A|I|F| mode
\end{verbatim}
Change mode (imod == 00 && M == 1)

CPS{<q>} #<mode> // (Not permitted in IT block)

Interrupt disable (imod == 11 && M == 0)

CPSID.W <iflags> // (Not permitted in IT block)

Interrupt disable and change mode (imod == 11 && M == 1)

CPS{<q>} <iflags>, #<mode> // (Not permitted in IT block)

Interrupt enable (imod == 10 && M == 0)

CPSIE.W <iflags> // (Not permitted in IT block)

Interrupt enable and change mode (imod == 10 && M == 1)

CPSIE{<q>} <iflags>, #<mode> // (Not permitted in IT block)

if imod == '00' && M == '0' then SEE "Hint instructions";
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10'); disable = (imod == '11'); changemode = (M == '1');
affectA = (A == '1'); affectI = (I == '1'); affectF = (F == '1');
if imod == '01' || InITBlock() then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If imod == '01', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If mode != '00000' && M == '0', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: changemode = TRUE.
- The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.

If imod<1> == '1' && A:I:F == '000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if imod<1> == '0'.
- The instruction behaves as if A:I:F has an UNKNOWN nonzero value.

If imod<1> == '0' && A:I:F != '000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if imod<1> == '1'.
- The instruction behaves as if A:I:F == '000'.

Hint instructions: In encoding T2, if the imod field is 00 and the M bit is 0, a hint instruction is encoded. To determine which hint instruction, see Branches and miscellaneous control.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.
<flags>
Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

\textbf{a}
Sets the A bit in the instruction, causing the specified effect on $PSTATE.A$, the SError interrupt mask bit.

\textbf{i}
Sets the I bit in the instruction, causing the specified effect on $PSTATE.I$, the IRQ interrupt mask bit.

\textbf{f}
Sets the F bit in the instruction, causing the specified effect on $PSTATE.F$, the FIQ interrupt mask bit.

<mode>
Is the number of the mode to change to, in the range 0 to 31, encoded in the "mode" field.

\textbf{Operation}

\begin{verbatim}
if CurrentInstrSet() == InstrSet_A32 then
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(mode);
    else
        EncodingSpecificOperations();
        if PSTATE.EL != EL0 then
            if enable then
                if affectA then PSTATE.A = '0';
                if affectI then PSTATE.I = '0';
                if affectF then PSTATE.F = '0';
            if disable then
                if affectA then PSTATE.A = '1';
                if affectI then PSTATE.I = '1';
                if affectF then PSTATE.F = '1';
            if changemode then
                // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
                AArch32.WriteModeByInstr(mode);
\end{verbatim}

CRC32

CRC32 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, or 32 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 \textbf{OPTIONAL} instruction, and in Armv8.1 it is mandatory for all implementations to implement it. \textbf{ID ISAR5} CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (\textbf{A1}) and T32 (\textbf{T1}).

\begin{verbatim}
A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 0 | 0 | 1 | 0 | sz | 0 | Rn | Rd | (0)(0)(0)(0) | 1 | 0 | 1 | 0 | Rm |
| cond | C |

CRC32B (sz == 00)

\texttt{CRC32B\{<q>\} <\texttt{Rd}>, <\texttt{Rn}>, <\texttt{Rm}>}

CRC32H (sz == 01)

\texttt{CRC32H\{<q>\} <\texttt{Rd}>, <\texttt{Rn}>, <\texttt{Rm}>}

CRC32W (sz == 10)

\texttt{CRC32W\{<q>\} <\texttt{Rd}>, <\texttt{Rn}>, <\texttt{Rm}>}

if ! \texttt{HaveCRCExt()} then UNDEFINED;
d = \texttt{UInt}(\texttt{Rd}); n = \texttt{UInt}(\texttt{Rn}); m = \texttt{UInt}(\texttt{Rm});
size = 8 \ll \texttt{UInt}(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;

\textbf{CONSTRAINED UNPREDICTABLE behavior}

If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32;

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

\textbf{T1}

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Rd</td>
<td>1</td>
<td>0</td>
<td>sz</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CRC32
CRC32B (sz == 00)
CRC32B{<q>} <Rd>, <Rn>, <Rm>

CRC32H (sz == 01)
CRC32H{<q>} <Rd>, <Rn>, <Rm>

CRC32W (sz == 10)
CRC32W{<q>} <Rd>, <Rn>, <Rm>

if InITBlock() then UNPREDICTABLE;
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32;.

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q> See Standard assembler syntax fields. An CRC32 instruction must be unconditional.

<Rd> Is the general-purpose accumulator output register, encoded in the "Rd" field.

<Rn> Is the general-purpose accumulator input register, encoded in the "Rn" field.

<Rm> Is the general-purpose data source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  acc = R[n]; // accumulator
  val = R[m]<size-1:0>; // input value
  poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
  tempacc = BitReverse(acc):Zeros(size);
  tempval = BitReverse(val):Zeros(32);
  // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
  R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**CRC32C**

CRC32C 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, or 32 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 ISARS**. CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 0 | 0 | 1 | 0 | sz | 0 | Rn | Rd | (0) | (0) | 1 | (0) | 0 | 1 | 0 | 0 | Rm
| cond | C |
```

**CRC32CB (sz == 00)**

CRC32CB\(<q>\) <Rd>, <Rn>, <Rm>

**CRC32CH (sz == 01)**

CRC32CH\(<q>\) <Rd>, <Rn>, <Rm>

**CRC32CW (sz == 10)**

CRC32CW\(<q>\) <Rd>, <Rn>, <Rm>

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32.

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

### T1

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | Rn | 1 | 1 | 1 | Rd | 1 | 0 | sz | Rm
| C |
```
CRC32CB (sz == 00)
CRC32CB(<q>, <Rd>, <Rn>, <Rm>)

CRC32CH (sz == 01)
CRC32CH(<q>, <Rd>, <Rn>, <Rm>)

CRC32CW (sz == 10)
CRC32CW(<q>, <Rd>, <Rn>, <Rm>)

if InITBlock() then UNPREDICTABLE;
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32.

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q> See Standard assembler syntax fields. An CRC32C instruction must be unconditional.
<Rd> Is the general-purpose accumulator output register, encoded in the "Rd" field.
<Rn> Is the general-purpose accumulator input register, encoded in the "Rn" field.
<Rm> Is the general-purpose data source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    acc = R[n];              // accumulator
    val = R[m]<size-1:0>;    // input value
    poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    tempacc = BitReverse(acc):Zeros(size);
    tempval = BitReverse(val):Zeros(32);
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction. No instruction other than branch instructions and instructions that write to the PC 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 and conditional instructions that write to the PC appearing in program order before the CSDB that have not been architecturally resolved.

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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

\[ \text{cond} \]

```
if cond \neq '1110' then UNPREDICTABLE; // CSDB must be encoded with AL condition
```

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{cond} \neq '1110' \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

### T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
```

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

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{InITBlock}() \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors.*
Assembler Symbols

<
See *Standard assembler syntax fields*.

<
See *Standard assembler syntax fields*.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    ConsumptionOfSpeculativeDataBarrier();
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
In Armv8, DBG executes as a NOP. Arm deprecates any use of the DBG instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

|cond| != 1111 | 0 0 1 1 0 | 0 1 0 | 0 0 0 | 0 | (1)(1)(1)(1)(0)(0)(0)(0) | 1 1 1 1 | option |

```
DBG{<c>}{<q>} #<option>
```

// DBG executes as a NOP. The 'option' field is ignored

**T1**

|cond| 1 1 1 0 0 1 1 0 | 1 0 | 0 | (1)(1)(1)(1)(0)(0)(0)(0) | 0 0 | 0 0 | 1 1 1 1 | option |

```
DBG{<c>}{<q>} #<option>
```

// DBG executes as a NOP. The 'option' field is ignored

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<option>` Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "option" field.

**Operation**

```c
if ConditionPassed() then
   EncodingSpecificOperations();
```

Internal version only: isa v01_19, pseudocode v2020-09 xml, sve v2020-09 rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DCPS1

Debug Change PE State to EL1 allows the debugger to move the PE into EL1 from EL0 or to a specific mode at the current Exception Level.

DCPS1 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is implemented, EL2 is implemented and enabled in the current Security state, and any of:
  - EL2 is using AArch32 and HCR.TGE is set to 1.
  - EL2 is using AArch64 and HCR_EL2.TGE is set to 1.

When the PE executes DCPS1 at EL0, EL1 or EL3:

- If EL3 or EL1 is using AArch32, the PE enters SVC mode and LR_svc, SPSR_svc, DLR, and DSPSR become UNKNOWN. If DCPS1 is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL1 is using AArch64, the PE enters EL1 using AArch64, selects SP_EL1, and ELR_EL1, ESR_EL1, SPSR_EL1, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

When the PE executes DCPS1 at EL2 the PE does not change mode, and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.

For more information on the operation of this instruction, see DCPS.

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 1 1 1 1 0 0 | 1 1 1 1 | 1 0 0 0 | 0 0 0 0 0 0 0 0 0 0 1

DCPS1

// No additional decoding required.
if \( \text{Halted}() \) then UNDEFINED;

if \( \text{EL2Enabled}() \) \&\& PSTATE.EL == \text{EL0} then
   \text{tg}e = if \( \text{ELUsingAArch32}() \) then HCR.TGE else HCR_EL2.TGE;
   if \( \text{tg}e \equiv '1' \) then UNDEFINED;

if PSTATE.EL != \text{EL0} || \( \text{ELUsingAArch32}() \) then
   if PSTATE.M == \text{M32 Monitor} then SCR.NS = '0';
   if PSTATE.EL != \text{EL0} \| \text{ELUsingAArch32}() then
      if PSTATE.E == \text{M32.EE} then
         AArch32.WRITE(M32_SVC);
         PSTATE.E = SCTLR.EE;
         if \( \text{HavePANExt}() \) \&\& SCTLR.SPAN == '0' then PSTATE.PAN = '1';
         LR_svc = bits(32) UNKNOWN;
         SPSR_svc = bits(32) UNKNOWN;
      else
         PSTATE.E = HSCTLR.EE;
         ELR_hyp = bits(32) UNKNOWN;
         HSR = bits(32) UNKNOWN;
         SPSR_hyp = bits(32) UNKNOWN;
      end
      DLR = bits(32) UNKNOWN;
      DSPSR = bits(32) UNKNOWN;
   else
      // Targeting EL1 using AArch64
      AArch64.MAYBEZEROREGISTERUPPERS();
      MaybeZeroSVEUpplers(EL1);
      PSTATE.nRW = '0';
      PSTATE.SP = '1';
      PSTATE.EL = \text{EL1};
      if \( \text{HavePANExt}() \) \&\& SCTLR_EL1.SPAN == '0' then PSTATE.PAN = '1';
      if \( \text{HaveUA0Ext}() \) then PSTATE.UAO = '0';
      ELR_EL1 = bits(64) UNKNOWN;
      ESR_EL1 = bits(64) UNKNOWN;
      SPSR_EL1 = bits(64) UNKNOWN;
      DLR_EL0 = bits(64) UNKNOWN;
      DSPSR_EL0 = bits(64) UNKNOWN;

      // SCTLR_EL1.IESB might be ignored in Debug state.
      if \( \text{HaveIESB}() \) \&\& SCTLR_EL1.IESB == '1' \&\& \( \text{ConstrainUnpredictableBool}() \) then
         SynchronizeErrors();
      UpdateEDSCRFields();
   end
end
DCPS2

Debug Change PE State to EL2 allows the debugger to move the PE into EL2 from a lower Exception level.

DCPS2 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is not implemented.
- The PE is in Secure state and any of:
  - Secure EL2 is not implemented.
  - Secure EL2 is implemented and Secure EL2 is disabled.

When the PE executes DCPS2:

- If EL2 is using AArch32, the PE enters Hyp mode and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.
- If EL2 is using AArch64, the PE enters EL2 using AArch64, selects SP_EL2, and ELR_EL2, ESR_EL2, SPSR_EL2, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of this instruction, see **DCPS**.

T1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

T1

DCPS2

if !HaveEL(EL2) then UNDEFINED;

Operation

if !Halted() || IsSecure() then UNDEFINED;

if ELUsingAArch32(EL2) then
  AArch32_WriteMode(M32_Hyp);
  PSTATE.E = HSCTRL.EE;
  ELR_hyp = bits(32) UNKNOWN;
  HSR = bits(32) UNKNOWN;
  SPSR_hyp = bits(32) UNKNOWN;
  DLR = bits(32) UNKNOWN;
  DSPSR = bits(32) UNKNOWN;
else                                        // Targeting EL2 using AArch64
  AArch64_MaybeZeroRegisterUppers();
  MaybeZeroSVEUppers(EL2);
  PSTATE.nRW = '0';
  PSTATE.SP = '1';
  PSTATE.EL = EL2;
  if HavePANExt() && SCTRL_EL2.SPAN == '0' && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' then
    PSTATE.PAN = '1';
  if HaveUAOExt() then PSTATE.UAO = '0';
  ELR_EL2 = bits(64) UNKNOWN;
  ESR_EL2 = bits(64) UNKNOWN;
  SPSR_EL2 = bits(64) UNKNOWN;
  DLR_EL0 = bits(64) UNKNOWN;
  DSPSR_EL0 = bits(64) UNKNOWN;
// SCTLR EL2.IESB might be ignored in Debug state.
if HaveIESB() && SCTRL_EL2.IESB == '1' && !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
  SynchronizeErrors();
UpdateEDSCRFields();                        // Update EDSCR PE state flags
DCPS3

Debug Change PE State to EL3 allows the debugger to move the PE into EL3 from a lower Exception Level or to a specific mode at the current Exception Level.

DCPS3 is undefined if any of:
- The PE is in Non-debug state.
- EL3 is not implemented.
- EDSCR.SDD is set to 1.

When the PE executes DCPS3:
- If EL3 is using AArch32, the PE enters Monitor mode and LR_mon, SPSR_mon, DLR and DSPSR become unknown. If DCPS3 is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL3 is using AArch64, the PE enters EL3 using AArch64, selects SP_EL3, and ELR_EL3, ESR_EL3, SPSR_EL3, DLR_EL0 and DSPSR_EL0 become unknown.

For more information on the operation of this instruction, see DCPS.

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 1 1 1 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1

T1

DCPS3

if !HaveEL(EL3) then UNDEFINED;
if !Halted() || EDSCR.SDD == '1' then UNDEFINED;

if ELUsingAArch32(EL3) then
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
            PSTATE.E = SCTLR.EE;

        LR mon = bits(32) UNKNOWN;
        SPSR_mon = bits(32) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
else // Targeting EL3 using AArch64
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL3);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL3;
    if HaveUAOExt() then PSTATE.UAO = '0';

    ELR_EL3 = bits(64) UNKNOWN;
    ESR_EL3 = bits(64) UNKNOWN;
    SPSR_EL3 = bits(64) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    sync errors = HaveIESB() && SCTLR_EL3.IESB == '1';
    if HaveDoubleFaultExt() && SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' then
        sync errors = TRUE;
    // SCTLR_EL3.IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync errors = FALSE;
    if sync errors then SynchronizeErrors();
UpdateEDSCRFields(); // Update EDSCR PE state flags
DMB

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

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
1 1 1 1 0 1 0 1 1 1 |(1)(1)(1)(1)(1)(1)(1)(1)(1)(0)(0)(0)(0) 0 1 0 1  option
```

// No additional decoding required

T1

```
1 1 1 1 0 0 1 1 1 0 1 |(1)(1)(1)(1)(1)(1) 0(0)(0)(1)(1)(1) 0 1 0 1  option
```

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>`: For encoding A1: see Standard assembler syntax fields. Must be AL or omitted.
  For encoding T1: see Standard assembler syntax fields.

- `<q>`: See Standard assembler syntax fields.

- `<option>`: Specifies an optional 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. Can be omitted. This option is referred to as the full system barrier. Encoded as option = 0b1111.
  - **ST**: Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. SYST is a synonym for ST. Encoded as option = 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 option = 0b1110.
  - **ISH**: Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b1011.
  - **ISHST**: Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 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 option = 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 option = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = 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 option = 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 option = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 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 option = 0b0001.

For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB). All other encodings of option are reserved. All unsupported and reserved options must execute as a full system DMB operation, but software must not rely on this behavior.

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

### Operation

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations();
  case option of
    when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
    when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
    when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
    when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
    when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
    when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
    when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
    when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
    when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
    when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
    when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
    otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;

  if PSTATE.EL IN {EL0, EL1} & EL2Enabled() then
    if HCR.BSU == '11' then
      domain = MBReqDomain_FullSystem;
    if HCR.BSU == '10' & domain != MBReqDomain_FullSystem then
      domain = MBReqDomain_OuterShareable;
    if HCR.BSU == '01' & domain == MBReqDomain_Nonshareable then
      domain = MBReqDomain_InnerShareable;
  DataMemoryBarrier(domain, types);
```
DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see Data Synchronization Barrier (DSB).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
</table>
| 1 1 1 1 0 1 0 1 0 1 1 1 (1)(1)(1)(1)(1)(1)(1)(0)(0)(0) 0 1 0 0 | \(\text{!= 0x00} \text{ option} \)

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
</table>
| 1 1 1 1 0 0 1 1 0 1 1 (1)(1)(1)(1) 1 0 (0) 0 (1)(1)(1)(1) 0 1 0 0 | \(\text{!= 0x00} \text{ option} \)

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields. Must be AL or omitted.

\(<q>\) For encoding T1: see Standard assembler syntax fields.

\(<\text{option}>\) Specifies an optional 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. Can be omitted. This option is referred to as the full system barrier. Encoded as option = 0b1111.

- **ST**: Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. SYST is a synonym for ST. Encoded as option = 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 option = 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 option = 0b1011.

- **ISHST**: Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 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 option = 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 option = 0b0111.

**NSHST**

Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = 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 option = 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 option = 0b0011.

**OSHST**

Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 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 option = 0b0001.

For more information on whether an access is before or after a barrier instruction, see *Data Synchronization Barrier (DSB)*. All other encodings of option are reserved, other than the values 0b0000 and 0b0100. All unsupported and reserved options must execute as a full system DSB operation, but software must not rely on this behavior.

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise
            if       option == '0000' then SEE "SSBB";
            elsif    option == '0100' then SEE "PSSBB";
            else     domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;
    end case;

if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
    if HCR.BSU == '11' then
        domain = MBReqDomain_FullSystem;
    else if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
        domain = MBReqDomain_OuterShareable;
    else if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
        domain = MBReqDomain_InnerShareable;
    end if;

DataSynchronizationBarrier(domain, types);

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
EOR, EORS (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.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccc}
\hline
!= & 1111 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & S & Rn & Rd & \text{imm12} \\
\end{array}
\]

cond

EOR (S == 0)

EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>

EORS (S == 1)

EORS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

\[
\begin{array}{cccccccccccccccc}
\text{d} & = & \text{UInt} & (\text{Rd}) ; & \text{n} & = & \text{UInt} & (\text{Rn}) ; & \text{setflags} & = & (\text{S} & = & '1'); \\
\text{(imm32, carry)} & = & \text{A32ExpandImm}_C & (\text{imm12}, \text{PSTATE}.C); \\
\end{array}
\]

T1

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 1 & 0 & i & 0 & 0 & 1 & 0 & 0 & S & \text{Rn} & 0 & \text{imm3} & \text{Rd} & \text{imm8} \\
\end{array}
\]

EOR (S == 0)

EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>

EORS (S == 1 && Rd != 1111)

EORS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

\[
\begin{array}{cccccccccccccccc}
\text{if Rd} & == & '1111' & \&\& & \text{S} & == & '1' & \text{then } \text{SEE } & \text{"TEQ (immediate)";} \\
\text{d} & = & \text{UInt} & (\text{Rd}) ; & \text{n} & = & \text{UInt} & (\text{Rn}) ; & \text{setflags} & = & (\text{S} & == & '1'); \\
\text{(imm32, carry)} & = & \text{T32ExpandImm}_C & (i:\text{imm3}:\text{imm8}, \text{PSTATE}.C); \\
\text{if } (\text{d} & == & 15 & \&\& & !\text{setflags}) & || & \text{n} & == & 15 & \text{then UNPREDICTABLE;} \\
\text{// Armv8-A removes UNPREDICTABLE for R13}
\end{array}
\]

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c>
See Standard assembler syntax fields.
</c>

<q>
See Standard assembler syntax fields.
</q>

<Rd>
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn>
For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const>
For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

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

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| != 1111 | 0 0 0 0 | 0 1 | S | Rn | Rd | imm5 | stype | 0 | Rm |
```

cond

**EOR, rotate right with extend (S == 0 & imm5 == 00000 & stype == 11)**

EOR{<c>{<q>}}{<Rd>,}{<Rn>, <Rm>, RRX}

**EOR, shift or rotate by value (S == 0 & !(imm5 == 00000 & stype == 11))**

EOR{<c>{<q>}}{<Rd>,}{<Rn>,<Rm>}{, <shift> #<amount>}

**EORS, rotate right with extend (S == 1 & imm5 == 00000 & stype == 11)**

EORS{<c>{<q>}}{<Rd>,}{<Rn>,<Rm>, RRX}

**EORS, shift or rotate by value (S == 1 & !(imm5 == 00000 & stype == 11))**

EORS{<c>{<q>}}{<Rd>,}{<Rn>,<Rm>}{, <shift> #<amount>}

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

### T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 1 0 0 0 0 | 0 0 0 1 | Rm | Rdn |
```

T1

EOR{<c>{<q>}}{<Rdn>,}{<Rn>,<Rm>} // (Inside IT block)

EORS{<c>{<Rdn>}}{<Rdn>,}{<Rn>,<Rm>} // (Outside IT block)

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```
EOR, rotate right with extend \((S == 0 & \& \text{imm3} == 000 & \& \text{imm2} == 00 & \& \text{stype} == 11)\)

\[
\text{EOR\{<c>\}\{<q>\} \{<Rd>,\} \{<Rn>, \{<Rm>, \{RRX}}
\]

EOR, shift or rotate by value \((S == 0 & \& !(\text{imm3} == 000 & \& \text{imm2} == 00 & \& \text{stype} == 11))\)

\[
\text{EOR\{<c>W\{<Rd>\},\} \{<Rn>, \{<Rm} \{, \{<shift> #<amount}>\}
\]

EORS, rotate right with extend \((S == 1 & \& \text{imm3} == 000 & \& \text{Rd} != 1111 & \& \text{imm2} == 00 & \& \text{stype} == 11)\)

\[
\text{EORS\{<c>\}\{<q>\} \{<Rd>,\} \{<Rn>, \{<Rm}, \{RRX}}
\]

EORS, shift or rotate by value \((S == 1 & \& !(\text{imm3} == 000 & \& \text{imm2} == 00 & \& \text{stype} == 11)) & \& \text{Rd} != 1111)\)

\[
\text{EORS.W\{<Rd>,\} \{<Rn>, \{<Rm} \{, \{<shift> #<amount}>\}
\]

For more information about the CONstrained UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>`: See Standard assembler syntax fields.
- `<q>`: See Standard assembler syntax fields.
- `<Rdn>`: Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.
- `<Rd>`: For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<Rn>\). Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  - For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
- `<Rn>`: For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
- `<Rm>`: For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<Rn>\).
- `<shift>`: Is the type of shift to be applied to the second source register, encoded in "stype":

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

```
if Rd == '1111' \&\& S == '1' then SEE "TEQ (register)";

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 \&\& !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if EORS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EORS <Rd>, <Rn> had been written.
- Inside an IT block, if EOR<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EOR<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    if d == 15 then  // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Bitwise Exclusive OR (register-shifted register) performs a bitwise Exclusive OR of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

### Flag setting (S == 1)

\[
\text{EORS}\{<c>\}<q>\{<Rd>,\} <Rn>, <Rm>, <\text{shift}> <Rs> \\
\]

### Not flag setting (S == 0)

\[
\text{EOR}\{<c>\}<q>\{<Rd>,\} <Rn>, <Rm>, <\text{shift}> <Rs> \\
\]

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<\text{shift}>` Is the type of shift to be applied to the second source register, encoded in "stype":

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

- `<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

---

EOR, EORS (register-shifted register)
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

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

Exception Return.
The PE branches to the address held in the register holding the preferred return address, and restores `PSTATE` from SPSR_<current_mode>.
The register holding the preferred return address is:

- `ELR_hyp`, when executing in Hyp mode.
- LR, when executing in a mode other than Hyp mode, User mode, or System mode.

The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.

Exception Return is **CONSTRAINED UNPREDICTABLE** in User mode and System mode.

In Debug state, the T1 encoding of ERET executes the DRPS operation.

It has encodings from the following instruction sets: A32 ( **A1** ) and T32 ( **T1** ).

**A1**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------------|
| != 1111 0 0 0 1 0 1 1 0 (0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0) |
```

A1

```
ERET{<c}<{q}>
```

// No additional decoding required

**T1**

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------------|
| 1 1 1 1 0 0 1 1 1 1 0 1 1 1 1 0 1 0 (0)(0) (1)(1)(1)(1) 0 0 0 0 0 0 0 0 0 0 0 0 0 |
```

T1

```
ERET{<c}<{q}>
```

if `InITBlock() & !LastInITBlock()` then UNPREDICTABLE;

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
else
    if Halted() then
        if PSTATE.M IN \{M32_User, M32_System\} then
            UNPREDICTABLE;                        // UNDEFINED or NOP
        else
            new_pc_value = if PSTATE.EL == EL2 then ELR_hyp else R[14];
            AArch32.ExceptionReturn(new_pc_value, SPSR[]);
        else
            UNDEFINED;
            // Perform DRPS operation in Debug state
        endif if PSTATE.M == M32_User then
            UNDEFINED;
        elseif PSTATE.M == M32_System then
            UNPREDICTABLE;                        // UNDEFINED or NOP
        else
            SynchronizeContext();
            bits(32) spsr = SPSR[];
            SetPSTATEFromPSR(spsr);
            // PSTATE.{N,Z,C,V,Q,GE,SS,A,I,F} are not observable and ignored in Debug state, so
            // behave as if UNKNOWN.
            PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
            // In AArch32 Debug state, all instructions are T32 and unconditional.
            PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
            DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
            UpdateEDSCRFields();                  // Update EDSCR PE state flags
        endif
    endif
endif

CONSTRANDED UNPREDICTABLE behavior

If PSTATE.M IN \{M32_User,M32_System\}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR and VDISR. 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.

If the RAS Extension is implemented, the encoding is as follows:

There are encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1
(Armv8.2)

```assembly
cond != 1111 | 0 0 1 1 0 | 0 0 0 0 0 | (1)(1)(1)(0)(0)(0) 0 0 0 0 1 0 0 0 0
```

A1

```assembly
ESB{<c>}{<q>}

if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
if cond != '1110' then UNPREDICTABLE; // ESB must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1
(Armv8.2)

```assembly
cond != 1111 | 0 0 1 1 0 | 0 0 0 0 0 | (1)(1)(1)(1)(0)(0)(0) 0 0 0 0 1 0 0 0 0
```

T1

```assembly
ESB{<c>}{<q>}

if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
if ConditionPassed() then
    EncodingSpecificOperations();
    SynchronizeErrors();
    AArch32.ESB0Operation();
    if PSTATE.EL IN {EL0, EL1} & EL2Enabled() then AArch32.vESB0Operation();
    TakeUnmaskedSErrorInterrupts();
**HLT**

Halting breakpoint causes a software breakpoint to occur. Halting breakpoint is always unconditional, even inside an IT block.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 0 0 1 0 0 0 0 imm12 0 1 1 1 imm4</td>
</tr>
</tbody>
</table>

**cond**

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 0 1 0 1 0 imm6</td>
</tr>
</tbody>
</table>

**Operation**

```
EncodingSpecificOperations();
Halt(DebugHalt_HaltInstruction);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Hypervisor Call causes a Hypervisor Call exception. For more information see \textit{Hypervisor Call (HVC) exception}. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service. The HVC instruction is:

- **UNDEFINED** in Secure state, and in User mode in Non-secure state.
- When \texttt{SCR.HCE} is set to 0, **UNDEFINED** in Non-secure EL1 modes and **CONSTRAINED UNPREDICTABLE** in Hyp mode.

On executing an HVC instruction, the \textit{HSR, Hyp Syndrome Register} reports the exception as a Hypervisor Call exception, using the EC value 0x12, and captures the value of the immediate argument, see \textit{Use of the HSR}.

It has encodings from the following instruction sets: A32 (\texttt{A1}) and T32 (\texttt{T1})

\textbf{A1}

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
| != 1111 | 0 0 0 1 0 1 0 0 | imm12
| cond

imm16 = imm12:imm4;
\end{verbatim}

\textbf{T1}

\begin{verbatim}
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
| 1 1 1 1 1 0 1 1 1 1 1 1 0 | imm4

imm16 = imm4:imm12;
if \texttt{InITBlock}() then UNPREDICTABLE;
\end{verbatim}

**CONSTRAINED UNPREDICTABLE behavior**

If \texttt{cond} != ‘1110’, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes unconditionally.
- The instruction executes conditionally.

\textbf{T1}

\begin{verbatim}
HVC{<q>} {#}<imm16>

imm16 = imm4:imm12;
if \texttt{InITBlock}() then UNPREDICTABLE;
\end{verbatim}

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see \textit{Architectural Constraints on UNPREDICTABLE behaviors}.

**Assembler Symbols**

\texttt{<q>}

See \textit{Standard assembler syntax fields}. An HVC instruction must be unconditional.

\texttt{<imm16>}

For encoding \texttt{A1}: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm12:imm4” field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

For encoding \texttt{T1}: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:imm12” field. This value is for assembly and disassembly only. It is reported in the HSR but
otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

**Operation**

```c
EncodingSpecificOperations();
if !HaveEL(EL2) || PSTATE.EL == EL0 || (IsSecure() && !IsSecureEL2Enabled()) then UNDEFINED;

if HaveEL(EL3) then
  if ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2 then UNPREDICTABLE;
  else
    hvc_enable = SCR_GEN[].HCE;
else
  hvc_enable = if ELUsingAArch32(EL2) then NOT(HCR.HCD) else NOT(HCR_EL2.HCD);

if hvc_enable == '0' then UNDEFINED;
else AArch32.CallHypervisor(imm16);
```

**CONSTRAINED UNPREDICTABLE behavior**

If ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see Instruction Synchronization Barrier (ISB).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 option</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 1 1 1 (1)(1)(1)(1)(1)(1)(1)(1)00011010</td>
</tr>
</tbody>
</table>

A1

ISB{<c>}{<q>} {<option>}

// No additional decoding required

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 option</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1 1 (1)(1)(1)(1)100011010</td>
</tr>
</tbody>
</table>

T1

ISB{<c>}{<q>} {<option>}

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. Must be AL or omitted.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

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

SY

Full system barrier operation, encoded as option = 0b1111. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
If-Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block are the same as, or the inverse of, the condition the IT instruction specifies for the first instruction in the block. The IT instruction itself does not affect the condition flags, but the execution of the instructions in the IT block can change the condition flags.

16-bit instructions in the IT block, other than CMP, CMN, and TST, do not set the condition flags. An IT instruction with the AL condition can change the behavior without conditional execution.

The architecture permits exception return to an instruction in the IT block only if the restoration of the CPSR restores PSTATE. IT to a state consistent with the conditions specified by the IT instruction. Any other exception return to an instruction in an IT block is unpredictable. Any branch to a target instruction in an IT block is not permitted, and if such a branch is made it is unpredictable what condition is used when executing that target instruction and any subsequent instruction in the IT block.

Many uses of the IT instruction are deprecated for performance reasons, and an implementation might include ITD controls that can disable those uses of IT, making them undefined.

For more information see Conditional execution and Conditional instructions. The first of these sections includes more information about the ITD controls.

```assembly
T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 1 1 1 1 1 firstcond != 0000

mask
```

IT{
{x}
{y}
{z}
}{q}
<cond>

if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;

CONSTRUANED UNPREDICTABLE behavior

If firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The '1111' condition is treated as being the same as the '1110' condition, meaning always, and the ITSTATE state machine is progressed in the same way as for any other cond_base value.

For more information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Miscellaneous 16-bit instructions.

Assembler Symbols

<x>

The condition for the second instruction in the IT block. If omitted, the "mask" field is set to 0b1000. If present it is encoded in the "mask[3]" field:

T

firstcond[0]

E

NOT firstcond[0]

<y>

The condition for the third instruction in the IT block. If omitted and <x> is present, the "mask[2:0]" field is set to 0b100. If <y> is present it is encoded in the "mask[2]" field:

T

firstcond[0]
The condition for the fourth instruction in the IT block. If omitted and `<y>` is present, the "mask[1:0]" field is set to 0b10. If `<z>` is present, the "mask[0]" field is set to 1, and it is encoded in the "mask[1]" field:

```
E  NOT firstcond[0]
```

The condition for the first instruction in the IT block, encoded in the "firstcond" field. See *Standard assembler syntax fields*.

The conditions specified in an IT instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check IT instruction syntax for validity but do not generate assembled instructions for them. See *Conditional instructions*.

**Operation**

```c
EncodingSpecificOperations();
AArch32.CheckITEnabled(mask);
PSTATE.IT<7:0> = firstcond:mask;
ShouldAdvanceIT = FALSE;
```

See *Standard assembler syntax fields*.

The condition for the first instruction in the IT block, encoded in the "firstcond" field. See *Condition codes* for the range of conditions available, and the encodings.

```
T  firstcond[0]
E  NOT firstcond[0]
```

See *Standard assembler syntax fields*.

The conditions specified in an IT instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check IT instruction syntax for validity but do not generate assembled instructions for them. See *Conditional instructions*.
LDA

Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
0 0 0 1 1 0 0 1 | Rn    | Rt  |

cond
```

```
LDA{<c>}{<q>}{<Rt>}, [<Rn>]

if t == 15 || n == 15 then UNPREDICTABLE;
```

**T1**

```
0 0 0 1 1 0 0 1 | Rn    | Rt  |
```

```
LDA{<c>}{<q>}{<Rt>}, [<Rn>]

if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

**Operation**

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = Mem0[address, 4];
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDAB**

Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | Rn | | | | | | | | | | | | | | | | | | | | | | | | | | | |

cond

```
A1

LDAB{<c>}{<q>}{<Rt>}, [<Rn>]

t = Uint(Rt); n = Uint(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | Rn | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

```
T1

LDAB{<c>}{<q>}{<Rt>}, [<Rn>]

t = Uint(Rt); n = Uint(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

**Operation**

```
if ConditionPassed() then
  EncodingSpecificOperations();
  address = R[n];
  R[t] = ZeroExtend(MemO[address, 1], 32);
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAEX

Load-Acquire Exclusive Word loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccccccccccccc}
\text{LDAEX}\{<c>\}\{<q>\} <Rt>, [<Rn>] & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & \text{Rn} & \text{Rt} & |(1)(1)(1)1010011(1)(1)(1)
\end{array}
\]

A1

\[
\begin{align*}
\text{LDAEX}\{<c>\}\{<q>\} <Rt>, [<Rn>] \\
t &= \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \\
\text{if } t == 15 || n == 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]

T1

\[
\begin{array}{cccccccccccccccccccccccccc}
\text{cond} & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\text{LDAEX}\{<c>\}\{<q>\} <Rt>, [<Rn>] & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & \text{Rn} & \text{Rt} & |(1)(1)(1)(1)111010(1)(1)(1)(1)
\end{array}
\]

T1

\[
\begin{align*}
\text{LDAEX}\{<c>\}\{<q>\} <Rt>, [<Rn>] \\
t &= \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \\
\text{if } t == 15 || n == 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

\[
\begin{align*}
<\text{c}> & \quad \text{See Standard assembler syntax fields.} \\
<\text{q}> & \quad \text{See Standard assembler syntax fields.} \\
<\text{Rt}> & \quad \text{Is the general-purpose register to be transferred, encoded in the “Rt” field.} \\
<\text{Rn}> & \quad \text{Is the general-purpose base register, encoded in the “Rn” field.}
\end{align*}
\]

Operation

\[
\begin{align*}
\text{if } \text{ConditionPassed}() \text{ then} \\
\quad \text{EncodingSpecificOperations();} \\
\quad \text{address} = R[n]; \\
\quad \text{AArch32.SetExclusiveMonitors(address, 4);} \\
\quad R[t] = \text{Mem0}[\text{address, 4}];
\end{align*}
\]

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAEXB

Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

• If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
• Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 0 0</td>
</tr>
</tbody>
</table>

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

**Operation**

if ConditionPassed() then

EncodingSpecificOperations();

address = R[n];

AArch32.SetExclusiveMonitors(address, 1);

R[t] = ZeroExtend(MemO[address, 1], 32);

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAEXD

Load-Acquire Exclusive Doubleword loads a doubleword from memory, writes it to two registers and:
  • If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor
  • Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also acts as a barrier instruction with the ordering requirements described in *Load-Acquire, Store-Release*.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```

\[
t = \text{UInt}(Rt); \quad t2 = t + 1; \quad n = \text{UInt}(Rn);
\]

if \(Rt<0> == '1' \text{ || } t2 == 15 \text{ || } n == 15\) then UNPREDICTABLE;

#### CONSTRAINED UNPREDICTABLE behavior

If \(Rt<0> == '1'\), then one of the following behaviors must occur:

  • The instruction is UNDEFINED.
  • The instruction executes as NOP.
  • The instruction executes with the additional decode: \(t<0> == '0'\).
  • The instruction executes with the additional decode: \(t2 = t\).
  • The instruction executes as described, with no change to its behavior and no additional side effects.

If \(Rt == '1110'\), then one of the following behaviors must occur:

  • The instruction is UNDEFINED.
  • The instruction executes as NOP.
  • The instruction is handled as described in *Using R15*.

### T1

```
LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
```

\[
t = \text{UInt}(Rt); \quad t2 = \text{UInt}(Rt2); \quad n = \text{UInt}(Rn);
\]

if \(t == 15 \text{ || } t2 == 15 \text{ || } t == t2 \text{ || } n == 15\) then UNPREDICTABLE;

#### CONSTRAINED UNPREDICTABLE behavior

If \(t == t2\), then one of the following behaviors must occur:

  • The instruction is UNDEFINED.
  • The instruction executes as NOP.
  • The load instruction executes but the destination register takes an UNKNOWN value.
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>`: See Standard assembler syntax fields.
- `<q>`: See Standard assembler syntax fields.
- `<Rt>`: For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. `<Rt>` must be even-numbered and not R14.
  For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rt2>`: For encoding A1: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`.
  For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Rn>`: Is the general-purpose base register, encoded in the "Rn" field.

Operation

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations();
  address = R[n];
  AArch32.SetExclusiveMonitors(address, 8);
  value = MemO[address, 8];
  // Extract words from 64-bit loaded value such that R[t] is
  // loaded from address and R[t2] from address+4.
  R[t] = if BigEndian(AccType_ORDERED) then value<63:32> else value<31:0>;
  R[t2] = if BigEndian(AccType_ORDERED) then value<31:0> else value<63:32>;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAEXH

Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in `Load-Acquire, Store-Release`. For more information about support for shared memory see `Synchronization and semaphores`. For information about memory accesses see `Memory accesses`.

It has encodings from the following instruction sets: A32 (`A1`) and T32 (`T1`).

**A1**

```
| ! = 1111 | 0 0 0 1 1 1 1 | Rn | Rt | (1)(1) 1 0 | 1 0 0 1 | (1)(1)(1)(1) |
```

cond

**T1**

```
| 1 1 1 1 0 1 0 0 1 1 0 1 | Rn | RT | (1)(1)(1)(1) 1 1 | 0 1 | (1)(1)(1)(1) |
```

**Assembler Symbols**

- `<c>` See `Standard assembler syntax fields`.
- `<q>` See `Standard assembler syntax fields`.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

**Operation**

- if `ConditionPassed()` then
  - EncodingSpecificOperations();
  - address = `R[n]`;
  - `AArch32.SetExclusiveMonitors(address, 2);`
  - `R[t] = ZeroExtend(MemO[address, 2], 32);`

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAH

Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

LDAH{<c>}{<q>} <Rt>, [<Rn>]

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

if \( t == 15 \) \( || \) \( n == 15 \) then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>Rn</td>
</tr>
</tbody>
</table>

LDAH{<c>}{<q>} <Rt>, [<Rn>]

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

if \( t == 15 \) \( || \) \( n == 15 \) then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\[<c>\] See Standard assembler syntax fields.

\[<q>\] See Standard assembler syntax fields.

\[<Rt>\] Is the general-purpose register to be transferred, encoded in the "Rt" field.

\[<Rn>\] Is the general-purpose base register, encoded in the "Rn" field.

Operation

\[
\text{if} \quad \text{ConditionPassed}() \quad \text{then}
\]

\[
\begin{align*}
\text{EncodingSpecificOperations}(); \\
\text{address} = R[n]; \\
R[t] = \text{ZeroExtend}((\text{MemO}[\text{address}, 2], 32));
\end{align*}
\]

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDC (immediate)

Load data to System register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to the DBGDTRXint System register. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses see Memory accesses.

In an implementation that includes EL2, the permitted LDC access to DBGDTRXint can be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see Trapping general Non-secure System register accesses to debug registers.

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>0</td>
<td>W</td>
<td>1</td>
<td>!= 1111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cond Rn

Offset (P == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>{, #<+-><imm>}]

Post-indexed (P == 0 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>], #<+-><imm>

Pre-indexed (P == 1 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>, #<+-><imm}>]

Unindexed (P == 0 && U == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>], <option>

if Rn == '1111' then SEE "LDC (literal)";
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 1111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>0</td>
<td>W</td>
<td>1</td>
<td>! = 1111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rn
Offset (P == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>{, #<+/-><imm>}]

Post-indexed (P == 0 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>], #<+/-><imm>

Pre-indexed (P == 1 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>], #<+/-><imm>]

Unindexed (P == 0 && U == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>], <option>

if Rn == '1111' then SEE “LDC (literal)";
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');

Assembler Symbols

<
See Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<Rn>
Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see LDC (literal).

<option>
Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the “imm8” field. The value of this field is ignored when executing this instruction.

+/−
Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

<imm>
Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the “imm8” field, as <imm>/4.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
  address = if index then offset_addr else R[n];

  // System register write to DBGDTRTXint.
  DBGDTR_EL0[] = MemA[address,4];
  if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDC (literal)

Load data to System register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to the **DBGDTRXT** System register. For information about memory accesses see Memory accesses.

In an implementation that includes EL2, the permitted LDC access to **DBGDTRXT** can be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see Trapping general Non-secure System register accesses to debug registers.

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

```
cond

A1 (!(P == 0 & & U == 0 & & W == 0))
```

LDC{<c>}{<q>} p14, c5, <label>
LDC{<c>}{<q>} p14, c5, [PC, #<imm>]
LDC{<c>}{<q>} p14, c5, [PC], <option>

if P == '0' & & U == '0' & & W == '0' then UNDEFINED;
index = (P == '1'); add = (U == '1'); cp = 14; imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' & & CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If W == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

```
T1 (!(P == 0 & & U == 0 & & W == 0))
```

LDC{<c>}{<q>} p14, c5, <label>
LDC{<c>}{<q>} p14, c5, [PC, #<imm>]

if P == '0' & & U == '0' & & W == '0' then UNDEFINED;
index = (P == '1'); add = (U == '1'); cp = 14; imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' & & CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If W == '1' || P == '0', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
• The instruction executes as LDC (immediate) with writeback to the PC. The instruction is handled as described in *Using R15*.

**Assembler Symbols**

:c: See *Standard assembler syntax fields*.
:q: See *Standard assembler syntax fields*.

<option> Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the “imm8” field. The value of this field is ignored when executing this instruction.

<label> The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE (encoded as U == 1). If the offset is negative, imm32 is equal to minus the offset and add == FALSE (encoded as U == 0).

+- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

<imm> Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the “imm8” field, as <imm>/4.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    address = if index then offset_addr else Align(PC,4);
    // System register write to DBGDTRTXint.
    DBGDTR_EL0[] = MemA[address,4];
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDM, LDMIA, LDMFD

Load Multiple (Increment After, Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. Related system instructions are LDM (User registers) and LDM (exception return).

This instruction is used by the alias POP (multiple registers).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

\[
\begin{array}{cccccccccccccccccccc}
\hline \\
!= & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & W & 1 & Rn & \text{register list} \\
\end{array}
\]

cond

A1

LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)

\[
n = \text{UInt}(Rn); \text{ registers = register list; wback = (W == '1');} \\
\text{if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;} \\
\text{if wback && registers<n> == '1' then UNPREDICTABLE;}
\]

CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If wback && registers<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

\[
\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline \\
1 & 1 & 0 & 0 & 1 & Rn & \text{register list} \\
\end{array}
\]

T1

LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)

\[
n = \text{UInt}(Rn); \text{ registers = '00000000':register list; wback = (registers<n> == '0');} \\
\text{if BitCount(registers) < 1 then UNPREDICTABLE;}
\]
CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

LDM{IA}{<c>}.W <Rn>!, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be represented in T1)
LDMF{<c>}.W <Rn>!, <registers> // (Alternate syntax, Full Descending stack, if <Rn>, '!' and <registers> can be represented in T1)
LDM{IA}{{<c>}{<q>}}<Rn>!, <registers> // (Preferred syntax)
LDMF{<c>}{<q>}<Rn>!, <registers> // (Alternate syntax, Full Descending stack)

n = UInt(Rn); registers = P:M:register_list; wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If wback && registers<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If BitCount(registers) == 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If registers<13> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If P == '1' && M == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.
For more information about the CONSTRANIED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

IA  Is an optional suffix for the Increment After form.

<c>  See Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<Rn>  Is the general-purpose base register, encoded in the "Rn" field.

For encoding A1 and T2: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

For encoding T1: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if <Rn> is included in <registers>, otherwise it must be present.

<registers>  For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R7, encoded in the "register_list" field.

For encoding T2: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP (multiple registers)</td>
<td>T2</td>
<td>W == '1' &amp;&amp; Rn == '1101' &amp;&amp; BitCount(P:M:register_list) &gt; 1</td>
</tr>
<tr>
<td>POP (multiple registers)</td>
<td>A1</td>
<td>W == '1' &amp;&amp; Rn == '1101' &amp;&amp; BitCount(register_list) &gt; 1</td>
</tr>
</tbody>
</table>

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4]; address = address + 4;
        if registers<15> == '1' then
            LoadWritePC(MemA[address,4]);
        if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
        if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDM (exception return)

Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The SPSR of the current mode is copied to the CPSR. An address adjusted by the size of the data loaded can optionally be written back to the base register. The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

Load Multiple (exception return) is:

- **UNDEFINED** in Hyp mode.
- **UNPREDICTABLE** in debug state, and in User mode and System mode.

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 1 | 0 | 0 | P | U | 1 | W | 1 | Rn | 1 | register_list |

cond

A1

LDM{<amode>}{<c>}{<q>}{Rn}{!}, <registers_with_pc>

n = UInt(Rn);  registers = register_list;
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If wback && registers<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all the loads using the specified addressing mode and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during the execution of this instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<amode> is one of:

- **DA**: Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
- **FA**: Full Ascending. For this instruction, a synonym for DA.
- **DB**: Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
- **EA**: Empty Ascending. For this instruction, a synonym for DB.
- **IA**: Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
- **FD**: Full Descending. For this instruction, a synonym for IA.
**IB**
Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

**ED**
Empty Descending. For this instruction, a synonym for IB.

See *Standard assembler syntax fields*.

See *Standard assembler syntax fields*.

<registers_with_pc> Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must be specified in the register list, and the instruction causes a branch to the address (data) loaded into the PC. See also *Encoding of lists of general-purpose registers and the PC*.

Instructions with similar syntax but without the PC included in the registers list are described in *LDM (User registers)*.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE_EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User, M32_System} then
        UNPREDICTABLE;
    else
        length = 4*BitCount(registers) + 4;
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;

        for i = 0 to 14
            if registers<i> == '1' then
                R[i] = MemA[address,4]; address = address + 4;
                new_pc_value = MemA[address,4];

        if wback && registers<n> == '0' then R[n] = if increment then R[n]+length else R[n]-length;
        if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

    AArch32.ExceptionReturn(new_pc_value, SPSR[]);
```

**CONSTRAINED UNPREDICTABLE behavior**

If PSTATE.M IN {M32_User, M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDM (User registers)

In an EL1 mode other than System mode, Load Multiple (User registers) loads multiple User mode registers from consecutive memory locations using an address from a base register. The registers loaded cannot include the PC. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Load Multiple (User registers) is UNDEFINED in Hyp mode, and UNPREDICTABLE in User and System modes.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC.

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
!= 1111 1 0 0 P U 1 (0) 1 Rn 0
```

cond

A1

```asm
LDM{<amode>}{<c>}{<q>} {Rn}, <registers_without_pc>
```

\[
n = \text{UInt}(Rn); \quad \text{registers} = \text{register_list}; \quad \text{increment} = (U == '1') \land \text{wordhigher} = (P == U);
\]

if \( n == 15 \lor \) BitCount(registers) < 1 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<amode>` is one of:
  - **DA**: Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as \( P = 0, U = 0 \).
  - **FA**: Full Ascending. For this instruction, a synonym for DA.
  - **DB**: Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as \( P = 1, U = 0 \).
  - **EA**: Empty Ascending. For this instruction, a synonym for DB.
  - **IA**: Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as \( P = 0, U = 1 \).
  - **FD**: Full Descending. For this instruction, a synonym for IA.
  - **IB**: Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as \( P = 1, U = 1 \).
Empty Descending. For this instruction, a synonym for IB.

See Standard assembler syntax fields.

See Standard assembler syntax fields.

Is the general-purpose base register, encoded in the "Rn" field.

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the LDM instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list. See also Encoding of lists of general-purpose registers and the PC.

Instructions with similar syntax but with the PC included in <registers_without_pc> are described in LDM (exception return).

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNDEFINED;
    elsif PSTATE.M IN {M32_User, M32_System} then UNPREDICTABLE;
    else
        length = 4*BitCount(registers);
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;
        for i = 0 to 14
            if registers<i> == '1' then // Load User mode register
                Rmode[i, M32_User] = MemA[address,4]; address = address + 4;

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User, M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDMDA, LDMFA

Load Multiple Decrement After (Full Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

Related system instructions are LDM (User registers) and LDM (exception return).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 1 | 0 | 0 | 0 | 0 | W | 1 | Rn | register_list |

cond

A1

LDMDA{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;

CONstrained unpREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If wback && registers<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONstrained unpREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4]; address = address + 4;
        if registers<15> == '1' then
            LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDMDB, LDMEA

Load Multiple Decrement Before (Empty Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the lowest of those locations can optionally be written back to the base register. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAO. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

Related system instructions are LDM (User registers) and LDM (exception return).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| != 1111 | 1 | 0 | 0 | 1 | 0 | 0 | W | 1 | Rn | register_list |

cond

LDMDB{<c>}{<q>}{<Rn>!}, <registers> // (Preferred syntax)

LDMEA{<c>}{<q>}{<Rn>!}, <registers> // (Alternate syntax, Empty Ascending stack)

n = UInt(Rn); registers = register_list; wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;

CONstrained unPredictable behavior

If wback && registers<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 1 1 0 1 0 0 | 1 | 0 | 0 | W | 1 | Rn | P | M | register_list |
LDMDB\{<c>\}{<q>} <Rn>\{!\}, <registers> // (Preferred syntax)

LDMEA\{<c>\}{<q>} <Rn>\{!\}, <registers> // (Alternate syntax, Empty Ascending stack)

\[n = \text{UInt}(Rn); \text{ registers} = P:M: \text{register list}; \text{ wback} = (W == '1');\]

if \(n == 15 \lor \text{BitCount}(\text{registers}) < 2 \lor (P == '1' \&\& M == '1')\) then UNPREDICTABLE;
if wback \&\& \text{ registers}<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' \&\& \text{ InITBlock}() \&\& !\text{LastInITBlock}() then UNPREDICTABLE;

CONstrained UnexpectedBehavior

If wback \&\& \text{ registers}<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If BitCount(registers) == 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If registers<13> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If P == '1' \&\& M == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For more information about the CONstrained UnexpectedBehavior behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rn>\) Is the general-purpose base register, encoded in the “Rn” field.

\(!\) The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

\(<\text{registers}>\) For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by \{ and \}. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list.
For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list:
- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

**Operation**

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            \( R[i] = \text{MemA}[\text{address},4]; \) address = address + 4;
        if registers<15> == '1' then
            LoadWritePC(MemA[address,4]);
        if wback && registers<n> == '0' then \( R[n] = R[n] - 4*\text{BitCount}(\text{registers}); \)
        if wback && registers<n> == '1' then \( R[n] = \text{bits(32) UNKNOWN}; \)
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDMIB, LDMED

Load Multiple Increment Before (Empty Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC. Related system instructions are LDM (User registers) and LDM (exception return).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------|--------|--------|--------|--------|--------|--------|
| != 1111                             | 1 0 0   | 1 1 0   | W      | Rn     |
| cond                                |        | register_list |

A1

LDMIB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
LDMED{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If wback && registers<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.
if \texttt{ConditionPassed}() then
  EncodingSpecificOperations();
  address = R[n] + 4;
  for i = 0 to 14
    if registers\langle i \rangle == '1' then
      R[i] = \texttt{MemA}[address,4];  address = address + 4;
    if registers\langle 15 \rangle == '1' then
      \texttt{LoadWritePC}(\texttt{MemA}[address,4]);
    if wback && registers\langle n \rangle == '0' then
      R[n] = R[n] + 4*\texttt{BitCount}(registers);
    if wback && registers\langle n \rangle == '1' then
      R[n] = \texttt{bits}(32) \texttt{UNKNOWN};

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

This instruction is used by the alias POP (single register).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3 and T4).

A1

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| != 1111          | 0 1 0           | P | U | 0 | W | 1 |
| cond             | Rt              | imm12 |
| Rn               |
```

Offset (P == 1 && W == 0)

LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm}>]

Post-indexed (P == 0 && W == 0)

LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

Pre-indexed (P == 1 && W == 1)

LDR{<c>}{<q>} <Rt>, [<Rn>, #(+/-)<imm>]

if Rn == '1111' then SEE "LDR (literal)";
if P == '0' && W == '1' then SEE "LDRT";

\[
t = \text{UInt}(Rt);
n = \text{UInt}(Rn);
imm32 = \text{ZeroExtend}(imm12, 32);
index = (P == '1');
add = (U == '1');
wback = (P == '0') || (W == '1');
\]

if wback && n == t then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|
| 0 1 1           | 0 1             | imm5           |
| Rn              | Rt              |
```

T1

LDR{<c>}{<q>} <Rt>, [<Rn> {, #<imm}>]

\[
t = \text{UInt}(Rt);
n = \text{UInt}(Rn);
imm32 = \text{ZeroExtend}(imm5:'00', 32);
index = TRUE;
add = TRUE;
wback = FALSE;
\]

T2

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|
| 1 0 0 1 1       | Rt              | imm8           |
```
T2

LDR{<c}<{q}>, [SP{, #{<imm>}}]

t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;

T3

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & = & 1111 & Rt & \text{imm12} \\
\end{array}
\]

Rn

T3

LDR{<c}.W <Rt>, [Rn} {, #<imm>} // (Rt, Rn, <imm> can be represented in T1 or T2)

LDR{<c}<{q}>, [Rn} {, #<imm}>]

if Rn == '1111' then SEE "LDR (literal)"

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & = & 1111 & Rt & \text{imm8} \\
\end{array}
\]

Rn

T4

Offset (P == 1 && U == 0 && W == 0)

LDR{<c}<{q}>, [Rn} {, #<imm>}]

Post-indexed (P == 0 && W == 1)

LDR{<c}<{q}>, [Rn}, #<imm>]

Pre-indexed (P == 1 && W == 1)

LDR{<c}<{q}>, [Rn}, #<imm>]

if Rn == '1111' then SEE "LDR (literal)"
if P == '1' && U == '1' && W == '0' then SEE "LDRT"
if P == '0' && W == '0' then UNDEFINED;

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & = & 1111 & Rt & \text{P | U | W | imm8} \\
\end{array}
\]

Rn

CONstrained UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

<Rn> For encoding A1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDR (literal).

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+/− Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.

For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.

For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP (single</td>
<td>A1 (post-indexed)</td>
<td>P == '0' &amp; U == '1' &amp; W == '0' &amp; Rn == '1101' &amp; imm12 == '000000000100'</td>
</tr>
<tr>
<td>register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POP (single</td>
<td>T4 (post-indexed)</td>
<td>Rn == '1101' &amp; P == '0' &amp; U == '1' &amp; W == '1' &amp; imm8 == '00000100'</td>
</tr>
<tr>
<td>register)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operation

if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
      if address<1:0> == '00' then
        LoadWritePC(data);
      else
        UNPREDICTABLE;
    else
      R[t] = data;
  else
    EncodingSpecificOperations();
    offset_addr = if add then (B[n] + imm32) else (B[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
      if address<1:0> == '00' then
        LoadWritePC(data);
      else
        UNPREDICTABLE;
    else
      R[t] = data;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 1 | 0 | P | U | 0 | W | 1 | 1 | 1 | 1 | 1 | Rt | imm12 |

cond

### A1 !(P == 0 & W == 1)

LDR{<c>}{<q>} <Rt>, <label> // (Normal form)

LDR{<c>}{<q>} <Rt>, [PC, #<+-<imm>] // (Alternative form)

if P == '0' & W == '1' then SEE "LDRT";

```
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32);
add = (U == '1'); wback = (P == '0') || (W == '1');
if wback then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDR (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 0 | 0 | 1 | Rt | imm8 |

### T1

LDR{<c>}{<q>} <Rt>, <label> // (Normal form)

```
t = UInt(Rt); imm32 = ZeroExtend(imm8:'00', 32); add = TRUE;
```

### T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### T2

LDR{<c>}.W <Rt>, <label> // (Preferred syntax, and <Rt>, <label> can be represented in T1)

LDR{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDR{<c>}{<q>} <Rt>, [PC, #<+-<imm>] // (Alternative syntax)

```
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); add = (U == '1');
if t == 15 & & InITBlock() & & !LastInITBlock() then UNPREDICTABLE;
```
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see `Architectural Constraints on UNPREDICTABLE behaviors`.

**Assemble Symbols**

<c>  
See [Standard assembler syntax fields](#).

<q>  
See [Standard assembler syntax fields](#).

<Rt>  
For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

<label>  
For encoding A1 and T2: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.

+/-  
Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

<imm>  
For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For encoding T2: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,4];
    if t == 15 then
        if address<1:0> == '00' then
            LoadWritePC(data);
        else
            UNPREDICTABLE;
    else
        R[t] = data;
```

**Operational information**

If CPSR.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. For information about memory accesses, see Memory accesses.

The T32 form of LDR (register) does not support register writeback.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

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

Offset (P == 1 & W == 0)

LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-.}<Rm>{, <shift>}}

Post-indexed (P == 0 & W == 0)

LDR{<c>}{<q>} <Rt>, [<Rn], {+/-.}<Rm>{, <shift>}}

Pre-indexed (P == 1 & W == 1)

LDR{<c>}{<q>} <Rt>, [<Rn], {+/-.}<Rm>{, <shift>}}

if P == '0' & W == '1' then SEE "LDRT";
  t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
  index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
  if m == 15 then UNPREDICTABLE;
  if wback && (n == 15 || n == t) then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If wback & n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
</tr>
</tbody>
</table>

T1

LDR{<c>}{<q>} <Rt>, [<Rn], {+.}<Rm>]

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRTType_LSL, 0);
LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]  
if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRTypelSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This branch is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.
For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.
For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/ Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.
if `CurrentInstrSet() == InstrSet_A32` then
    if `ConditionPassed()` then
        EncodingSpecificOperations();
        offset = `Shift(R[m], shift_t, shift_n, PSTATE.C)`;
        offset_addr = if add then `(R[n] + offset)` else `(R[n] - offset)`;
        address = if index then offset_addr else `R[n]`;
        data = `MemU[address,4]`;
        if wback then `R[n] = offset_addr`;
        if t == 15 then
            if address<1:0> == '00' then
                `LoadWritePC(data)`;
            else
                UNPREDICTABLE;
            else
                `R[t] = data`;
    else
        if `ConditionPassed()` then
            EncodingSpecificOperations();
            offset = `Shift(R[m], shift_t, shift_n, PSTATE.C)`;
            offset_addr = `(R[n] + offset)`;
            address = offset_addr;
            data = `MemU[address,4]`;
            if t == 15 then
                if address<1:0> == '00' then
                    `LoadWritePC(data)`;
                else
                    UNPREDICTABLE;
            else
                `R[t] = data`;

Operational information

If CPSR.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) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 1 | 0 | P | U | 1 | W | 1 | != 1111 | Rt | imm12 |
| cond | Rn |

Offset (P == 1 && W == 0)

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #/{-}/<imm>}]

Post-indexed (P == 0 && W == 0)

LDRB{<c>}{<q>} <Rt>, [<Rn]>, #/{-}/<imm>

Pre-indexed (P == 1 && W == 1)

LDRB{<c>}{<q>} <Rt>, [<Rn>], #/{-}/<imm>

if Rn == '1111' then SEE "LDRB (literal)";
if P == '0' && W == '1' then SEE "LDRBT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If wback & n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 1 | 1 | 1 | imm5 | Rn | Rt |

T1

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #/{+}/<imm>}]

t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;

T2

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | != 1111 | ! == 1111 | imm12 |
| Rn | Rt |
LDRB{<c>}.W <Rt>, [<Rn> {, #(+)<imm}>] // (<Rt>, <Rn>, <imm> can be represented in T1)

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #(+)<imm}>]

if Rt == "1111" then SEE "PLD";
if Rn == "1111" then SEE "LDRB (literal)";
  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
  index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13

T3

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 0 0 0 0 0 0 0 1 != 1111  Rt  1  P  U  W  imm8
Rn

Offset (Rt != 1111 \&\& P == 1 \&\& U == 0 \&\& W == 0)

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm}>]

Post-indexed (P == 0 \&\& W == 1)

LDRB{<c>}{<q>} <Rt>, [<Rn>], #{/-}<imm>

Pre-indexed (P == 1 \&\& W == 1)

LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/}<imm>

if Rt == '1111' \&\& P == '1' \&\& U == '0' \&\& W == '0' then SEE "PLD, PLDW (immediate)";
if Rn == '1111' then SEE "LDRB (literal)";
if P == '1' \&\& U == '1' \&\& W == '0' then SEE "LDRBT";
if P == '0' \&\& W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 \&\& W == '1') || (wback \&\& n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13

CONCONSTRAINED UNPREDICTABLE behavior

If wback \&\& n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONCONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>  See Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<Rt>  Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn>  For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRB (literal).

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.
Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the “imm5” field.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

**Operation**

```plaintext
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
      address = if index then offset_addr else R[n];
      R[t] = ZeroExtend(MemU[address,1], 32);
      if wback then R[n] = offset_addr;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{ccccccccccccccccccccc}
\hline
!= 1111 & 0 & 1 & 0 & P & U & 1 & W & 1 & 1 & 1 & 1 & Rt & \text{imm12} \\
\end{array}
\]

\text{cond}

A1 !(P == 0 && W == 1))

LDRB{<c>}{<q>} <Rt>, <label> // (Normal form)
LDRB{<c>}{<q>} <Rt>, [PC, #<imm>] // (Alternative form)

if P == '0' && W == '1' then SEE "LDRBT";
    t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);
    add = (U == '1');  wback = (P == '0') || (W == '1');
    if t == 15 || wback then UNPREDICTABLE;

CONSTRUDED UNPREDICTABLE behavior

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRB (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1

\[
\begin{array}{ccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & != 1111 & \text{imm12} & \text{Rt} \\
\end{array}
\]

T1

LDRB{<c>}{<q>} <Rt>, <label> // (Preferred syntax)
LDRB{<c>}{<q>} <Rt>, [PC, #<imm>] // (Alternative syntax)

if Rt == '1111' then SEE "PLD";
    t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
    // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRUDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<\text{c}> \quad \text{See Standard assembler syntax fields.}
<\text{q}> \quad \text{See Standard assembler syntax fields.}
<\text{Rt}> \quad \text{Is the general-purpose register to be transferred, encoded in the "Rt" field.}
<label> The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.
If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/− Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th>U</th>
<th>+/−</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC, 4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|
| != 1111 | 0 1 1 | P | U | 1 | W | 1 | Rn | Rt | imm5 | stype | 0 | Rm |
| cond

Offset (P == 1 & W == 0)

LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]}

Post-indexed (P == 0 & W == 0)

LDRB{<c>}{<q>} <Rt>, [<Rn]}, {+/-}<Rm>{, <shift>}

Pre-indexed (P == 1 & W == 1)

LDRB{<c>}{<q>} <Rt>, [<Rn], {+/-}<Rm>{, <shift}>}!

if P == ’0’ & W == ‘1’ then SEE “LDRBT”;

if P == ’0’ & W == ‘1’ then SEE “LDRBT”;

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == ‘1’);  add = (U == ‘1’);  wback = (P == ‘0’) || (W == ‘1’);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 1 1 | 1 | 0 | Rm | Rn | Rt |

T1

LDRB{<c>}{<q>} <Rt>, [<Rn], {+}<Rm>]

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
LDRB\{<c>\}.W \{<Rt>, [<Rn>, {+}<Rm>] \} // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRB\{<c>\}\{<q>\} \{<Rt>, [<Rn>, {+}<Rm>\}, LSL #<imm>\}

if Rt = '1111' then SEE "PLD";
if Rn = '1111' then SEE "LDRB (literal)"

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wbback = FALSE;
(shift_t, shift_n) = (SRType.LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.
\(<Rn>\) For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.
For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.
\(+/-\) Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

\(+\) Specifies the index register is added to the base register.

\(<Rm>\) Is the general-purpose index register, encoded in the "Rm" field.

\(<shift>\) The shift to apply to the value read from \(<Rm>\). If absent, no shift is applied. Otherwise, see Shifts applied to a register.

\(<imm>\) If present, the size of the left shift to apply to the value from \(<Rm>\), in the range 1-3. \(<imm>\) is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  address = if index then offset_addr else R[n];
  R[t] = ZeroExtend(MemU[address,1],32);
  if wbback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRBT

Load Register Byte

Unprivileged loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRBT is *UNPREDICTABLE* in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (*A1* and *A2*) and T32 (*T1*).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 0 | 1 | 0 | 0 | U | 1 | 1 | 1 | Rn | Rt | imm12 |
| cond |

**A1**

```assembly
LDRBT{<c>}{<q>}{<Rt>}, [<Rn>] {, #{+/−}<imm>}
```

```c
int t = UInt(Rt); int n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If `n == 15`, then one of the following behaviors must occur:

- The instruction is *UNDEFINED*.
- The instruction executes as *NOP*.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is *UNDEFINED*.
- The instruction executes as *NOP*.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is *UNKNOWN*. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

**A2**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 0 | 1 | 1 | 0 | U | 1 | 1 | 1 | Rn | Rt | imm5 | stype | 0 | Rm |
| cond |

**A2**

```assembly
LDRBT{<c>}{<q>}{<Rt>}, [<Rn>], {+/−}<Rm>{, <shift>}
```

```c
int t = UInt(Rt); int n = UInt(Rn); int m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE; (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```
CONSTRANIED UNPREDICTABLE behavior

If n == t && n != 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Rn</th>
<th>imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 0 0 0 0 1 != 1111</td>
<td>Rt</td>
<td>1 1 1 0</td>
</tr>
</tbody>
</table>

LDRBT{<c>}{<q>}{<Rt>, [<Rn> {, #{+}<imm>}]}

if Rn == '1111' then SEE "LDRB (literal)";
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRANIED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- <c> See Standard assembler syntax fields.
- <q> See Standard assembler syntax fields.
- <Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.
- +/- For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rm> Is the general-purpose index register, encoded in the "Rm" field.
- <shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register.
- + Specifies the offset is added to the base register.
- <imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

LDRBT
Operation

if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = ZeroExtend(MemU_unpriv[address,1],32);
    if postindex then R[n] = offset_addr;

CONstrained UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRB (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | P | U | 1 | W | 0 | != | 1111 | Rt | imm4H | 1 | 1 | 0 | 1 | imm4L |
| cond | Rn |

Offset (P == 1 & W == 0)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #(+/-)<imm}>]

Post-indexed (P == 0 & W == 0)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #(+/-)<imm>

Pre-indexed (P == 1 & W == 1)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #(+/-)<imm>!

if Rn == '1111' then SEE "LDRD (literal)";
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt); t2 = t+1; n = UInt(Rn); imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
if P == '0' & W == '1' then UNPREDICTABLE;
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If wback && (n == t || n == t2), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If P == '0' & W == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If Rt<0> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

T1
Offset (P == 1 && W == 0)

LDRD<
\{<q> \} 
\{<Rt>, <Rt2>, [<Rn> \{, \#\{+/-\}\<imm>\}]\}

Post-indexed (P == 0 && W == 1)

LDRD<
\{<q> \} 
\{<Rt>, <Rt2>, [<Rn>], \#\{+/-\}\<imm>\}

Pre-indexed (P == 1 && W == 1)

LDRD<
\{<q> \} 
\{<Rt>, <Rt2>, [<Rn>], \#\{+/-\}\<imm>\}!

if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE "LDRD (literal)";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If wback & (n == t || n == t2), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Load/store dual, load/store exclusive, table branch.

Assembler Symbols

<\c> See Standard assembler syntax fields.

<\q> See Standard assembler syntax fields.

<\Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

<\Rt2> For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<\Rn> Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRD (literal).

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>\U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>
For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as \(<imm>/4\).

**Operation**

```assembly
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccType_ATOMIC) then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // (Normal form)
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #<+/-><imm>] // (Alternative form)

if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t2 == 15 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If Rt<0> == '1', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes as if P == 1 and W == 0.'

T1

LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // (Normal form)
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #<+/-><imm>] // (Alternative form)

if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if W == '1' then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If t == t2, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The load instruction executes but the destination register takes an \textit{UNKNOWN} value.

If W == '1', then one of the following behaviors must occur:

• The instruction is \textit{UNDEFINED}.
• The instruction executes as \textit{NOP}.
• The instruction executes without writeback of the base address.
• The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in \textit{Using R15}.

For more information about the \textit{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see \textit{Architectural Constraints on UNPREDICTABLE behaviors}.

Related encodings: \textit{Load/Store dual, Load/Store-Exclusive, Load-Acquire/Store-Release, table branch}.

\textbf{Assembler Symbols}

\textbf{<c> See Standard assembler syntax fields.}

\textbf{<q> See Standard assembler syntax fields.}

\textbf{<Rt>} For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

\textbf{<Rt2>} For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

\textbf{<label>} For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

\textbf{+/-} Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

\begin{center}
\begin{tabular}{c|c}
U & +/- \\
0 & - \\
1 & +
\end{tabular}
\end{center}

\textbf{<imm>} For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see \textit{Use of labels in UAL instruction syntax}. 

\textbf{LDRD (literal)}
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccType_ATOMIC) then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRD (register)

Load Register Dual (register) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

A1

<table>
<thead>
<tr>
<th>!= 1111</th>
<th>0 0 0</th>
<th>P</th>
<th>U</th>
<th>0</th>
<th>W</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
<th>(0)(0)(0)(0)</th>
<th>1 1 0 1</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset (P == 1 && W == 0)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]

Post-indexed (P == 0 && W == 0)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>

Pre-indexed (P == 1 && W == 1)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]

if Rt<0> == '1' then UNPREDICTABLE:
  t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
  index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
  if P == '0' && W == '1' then UNPREDICTABLE;
  if t2 == 15 || m == 15 || m == t || m == t2 then UNPREDICTABLE;
  if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If wback && (n == t || n == t2), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

If P == '0' && W == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If m == t || m == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads register Rm with an UNKNOWN value.

If Rt<0> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

LDRD (register)
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;c&gt;</td>
<td>See Standard assembler syntax fields.</td>
</tr>
<tr>
<td>&lt;q&gt;</td>
<td>See Standard assembler syntax fields.</td>
</tr>
<tr>
<td>&lt;Rt&gt;</td>
<td>Is the first general-purpose register to be transferred, encoded in the &quot;Rt&quot; field. This register must be even-numbered and not R14.</td>
</tr>
<tr>
<td>&lt;Rt2&gt;</td>
<td>Is the second general-purpose register to be transferred. This register must be &lt;R(t+1)&gt;.</td>
</tr>
<tr>
<td>&lt;Rn&gt;</td>
<td>Is the general-purpose base register, encoded in the &quot;Rn&quot; field. The PC can be used in the offset variant.</td>
</tr>
<tr>
<td>+/-</td>
<td>Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in &quot;U&quot;:</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>&lt;Rm&gt;</td>
<td>Is the general-purpose index register, encoded in the &quot;Rm&quot; field.</td>
</tr>
</tbody>
</table>

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccType_ATOMIC) then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDREX

Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```assembly
LDREX{<c>}{<q>}{<Rt>}, [<Rn> {, {#}<imm>}]
```

```assembly
t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if t == 15 || n == 15 then UNPREDICTABLE;
```

### T1

```assembly
LDREX{<c>}{<q>}{<Rt>}, [<Rn> {, #<imm>}]
```

```assembly
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
- `<imm>` For encoding A1: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can only be 0 or omitted.
  
  For encoding T1: the immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

**Operation**

```assembly
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    AArch32_SetExclusiveMonitors(address, 4);
    R[t] = MemA[address, 4];
```

LDREX
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDREXB

Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

• If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
• Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | Rn | | | | | | | | | | | | | | | | | | | | | |
| cond |

A1

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | Rn | | | | | | | | | | | | | | | | | | | | | |

T1

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- **<c>** See *Standard assembler syntax fields*.
- **<q>** See *Standard assembler syntax fields*.
- **<Rt>** Is the general-purpose register to be transferred, encoded in the “Rt” field.
- **<Rn>** Is the general-purpose base register, encoded in the “Rn” field.

**Operation**

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32_SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDREXD

Load Register Exclusive Doubleword derives an address from a base register value, loads a 64-bit doubleword from memory, writes it to two registers and:

• If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
• Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | Rn | Rt | (1) | (1) | 1 | 1 | 1 | 0 | 0 | 1 | (1) | (1) | (1) | (1) |

cond

A1

LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

t = UInt(Rt);  t2 = t + 1;  n = UInt(Rn);
if Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If Rt<0> == '1', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes with the additional decode: t<0> = '0'.
• The instruction executes with the additional decode: t2 = t.
• The instruction executes as described, with no change to its behavior and no additional side effects.

If Rt == '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction is handled as described in *Using R15*.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | Rn | Rt | Rt2 | 0 | 1 | 1 | (1) | (1) | (1) |

T1

LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If t == t2, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The load instruction executes but the destination register takes an UNKNOWN value.
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. `<Rt>` must be even-numbered and not R14.
  
  For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rt2>` For encoding A1: is the second general-purpose register to be transferred. `<Rt2>` must be `<R(t+1)>`.
  
  For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,8);
    value = MemA[address,8];
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R[t] = if BigEndian(AccType_ATOMIC) then value<63:32> else value<31:0>;
    R[t2] = if BigEndian(AccType_ATOMIC) then value<31:0> else value<63:32>;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDREXH**

Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

cond

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]  

\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \]
\[ \text{if } t == 15 \text{ || } n == 15 \text{ then UNPREDICTABLE; } \]
```

### T1

```
|    | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
15 | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  |
```

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]  

\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \]
\[ \text{if } t == 15 \text{ || } n == 15 \text{ then UNPREDICTABLE; } \quad \text{// Armv8-A removes UNPREDICTABLE for R13} \]
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the “Rt” field.
- `<Rn>` Is the general-purpose base register, encoded in the “Rn” field.

### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2],32);
```

### Operational information

If CPSR.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) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|----------------|----------------|
| != 1111 | 0 0 0 | P | U | 1 | W | 1 | != 1111 | Rt | imm4H | 1 | 0 | 1 | 1 | imm4L |
| cond | Rn |

Offset (P == 1 && W == 0)

LDRH{<c>}{<q>} <Rt>, [<Rn> {, #(+/-)<imm}>]

Post-indexed (P == 0 && W == 0)

LDRH{<c>}{<q>} <Rt>, [<Rn>], #(+/-)<imm>

Pre-indexed (P == 1 && W == 1)

LDRH{<c>}{<q>} <Rt>, [<Rn>], #(+/-)<imm>!

if Rn == '1111' then SEE "LDRH (literal)";
if P == '0' && W == '1' then SEE "LDRHT";
* t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
it t == 15 || (wback & n == t) then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If wback & n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|----------------|
| 0 0 0 0 1 | imm5 | Rn |

T1

LDRH{<c>}{<q>} <Rt>, [<Rn> {, #(+)<imm}>]

* t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:0, 32);
index = TRUE;  add = TRUE;  wback = FALSE;

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|----------------|
| 1 1 1 1 0 0 0 0 1 0 1 1 | != 1111 | != 1111 | imm12 |

| Rn | Rt |
T2

LDRH{{<c>}.W <Rt>, [<Rn> {, #(+)<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)

LDRH{{<c>}{<q>} <Rt>, [<Rn> {, #(+)<imm}>]}

if Rt == '1111' then SEE "PLD (immediate)";
if Rn == '1111' then SEE "LDRH (literal)"

\[
t = \text{UINT}(Rt);
\]
\[
n = \text{UINT}(Rn);
\]
\[
\text{imm32} = \text{ZeroExtend} (\text{imm12, 32});
\]
\[
\text{index} = \text{TRUE};
\]
\[
\text{add} = \text{TRUE};
\]
\[
\text{wback} = \text{FALSE};
\]

// Armv8-A removes UNPREDICTABLE for R13

T3

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1111 != Rt

Offset (Rt != 1111 & P == 1 & U == 0 & W == 0)

LDRH{{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]}

Post-indexed (P == 0 & W == 1)

LDRH{{<c>}{<q>} <Rt>, [<Rn>], #(-)<imm>}

Pre-indexed (P == 1 & W == 1)

LDRH{{<c>}{<q>} <Rt>, [<Rn>], #(+)<imm>!}

if Rn == '1111' then SEE "LDRH (literal)"
if Rt == '1111' & P == '1' & U == '0' & W == '0' then SEE "PLDW (immediate)"
if P == '1' & U == '1' & W == '0' then SEE "LDRHT"
if P == '0' & W == '0' then UNDEFINED;

\[
t = \text{UINT}(Rt);
\]
\[
n = \text{UINT}(Rn);
\]
\[
\text{imm32} = \text{ZeroExtend} (\text{imm8, 32});
\]
\[
\text{index} = (P == '1');
\]
\[
\text{add} = (U == '1');
\]
\[
\text{wback} = (W == '1');
\]

// Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If \( wback \) && \( n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>`  See Standard assembler syntax fields.
- `<q>`  See Standard assembler syntax fields.
- `<Rt>`  Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>`  For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRH (literal).

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.
Specifies the offset is added to or subtracted from the base register, defaulting to $+$ if omitted and encoded in “$U$”:

<table>
<thead>
<tr>
<th>$U$</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

Specifies the offset is added to the base register.

For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the “imm5” field as $<$imm$/2$.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

**Operation**

```plaintext
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,2];
        if wback then R[n] = offset_addr;
        R[t] = ZeroExtend(data, 32);
    else
        if ConditionPassed() then
            EncodingSpecificOperations();
            offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
            address = if index then offset_addr else R[n];
            data = MemU[address,2];
            if wback then R[n] = offset_addr;
            R[t] = ZeroExtend(data, 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|----------------------------------------|----------------------------------------|----------------------------------------|
| != 1111                               | 0 0 0 | P | U | 1 | W | 1 | 1 | 1 | 1 | 1 | 1 |       | Rt | imm4H | 1 | 0 | 1 | 1 | imm4L |
| cond                                  |                                  |                                  |                                  |

A1 (!(P == 0 && W == 1))

LDRH{<c>}{<q>} <Rt>, <label> // (Normal form)
LDRH{<c>}{<q>} <Rt>, [PC, #<imm>] // (Alternative form)

if P == '0' && W == '1' then SEE "LDRHT";
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt); imm32 = ZeroExtend(imm4H:imm4L, 32);
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32);
add = (U == '1'); wback = (P == '0') || (W == '1');
add = (U == '1'); wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;

CONstrained Unpredictable behavior

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRH (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|----------------------------------------|----------------------------------------|----------------------------------------|
| 1 1 1 1 1 1 0 0 0 | U | 0 | 1 | 1 | 1 | 1 | 1 | != 1111 | imm12 |
| Rt                                  |                                  |                                  |                                  |

T1

LDRH{<c>}{<q>} <Rt>, <label> // (Preferred syntax)
LDRH{<c>}{<q>} <Rt>, [PC, #<imm>] // (Alternative syntax)

if Rt == '1111' then SEE "PLD (literal)"
if Rt == '1111' then SEE "PLD (literal)"
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13

// Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
<label> For encoding A1: the label of the literal data item that is to be loaded into &lt;Rt&gt;. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.
If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.
For encoding T1: the label of the literal data item that is to be loaded into &lt;Rt&gt;. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.
If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC, 4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = ZeroExtend(data, 32);

Operational information

If CPSR.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 to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0 | 0 | 0 | P | U | 0 | W | 1 | Rn | Rt | (0) | (0) | (0) | (0) | 1 | 0 | 1 | 1 | Rm |

cond

Offset (P == 1 && W == 0)

LDRH{<c>{<q>}{<Rt>, [<Rn>, {+/-}<Rm>]

Post-indexed (P == 0 && W == 0)

LDRH{<c>{<q>}{<Rt>, [<Rn]}, {+/-}<Rm>

Pre-indexed (P == 1 && W == 1)

LDRH{<c>{<q>}{<Rt>, [<Rn], {+/-}<Rm}!

if P == '0' && W == '1' then SEE "LDRHT";
  t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
  index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
  (shift_t, shift_n) = (SRType_LSL, 0);
  if t == 15 || m == 15 then UNPREDICTABLE;
  if wback && n == t then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 0 | 1 | 0 | 1 | Rm | Rn | Rt |

T1

LDRH{<c>{<q>}{<Rt>, [<Rn], {+}<Rm}]

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}}

if Rn == '1111' then SEE "LDRH (literal)";
if Rt == '1111' then SEE "PLDW (register)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<
See Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<Rt>
Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn>
For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.
For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/-
Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U +/- 0 -
1 +

+ Specifies the index register is added to the base register.

<Rm>
Is the general-purpose index register, encoded in the "Rm" field.

<imm>
If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

if ConditionPassed() then
   EncodingSpecificOperations();
   offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
   offset_addr = if add then (R[n] + offset) else (R[n] - offset);
   address = if index then offset_addr else R[n];
   data = MemU(address,2);
   if wback then R[n] = offset_addr;
   R[t] = ZeroExtend(data, 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDRHT

Load Register Halfword Unprivileged loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 0 0 0 U 1 1 1 Rn Rt imm4H 1 0 1 1 imm4L</td>
</tr>
</tbody>
</table>

A1

LDRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/−}<imm>}

\[
\begin{align*}
& t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad \text{postindex} = \text{TRUE}; \quad \text{add} = (U == '1'); \\
& \text{register form} = \text{FALSE}; \quad \text{imm32} = \text{ZeroExtend}(\text{imm4H}:\text{imm4L}, 32); \\
& \text{if } t == 15 || n == 15 || n == t \text{ then UNPREDICTABLE};
\end{align*}
\]

CONstrained unpredictablE behavior

If \( n == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If \( n == t \) \&\& \( n != 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 0 0 0 U 0 1 1 Rn Rt (0)(0)(0)(0) 1 0 1 1 Rm</td>
</tr>
</tbody>
</table>

A2

LDRHT{<c>}{<q>} <Rt>, [<Rn]>, {+/−}<Rm>

\[
\begin{align*}
& t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{postindex} = \text{TRUE}; \quad \text{add} = (U == '1'); \\
& \text{register form} = \text{TRUE}; \\
& \text{if } t == 15 || n == 15 || n == t || m == 15 \text{ then UNPREDICTABLE};
\end{align*}
\]
CONSTRANDED UNPREDICTABLE behavior

If \( n = t \) & \( n \neq 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
\]

if \( Rn = '1111' \) then SEE "LDRH (literal)"

\[
\begin{align*}
t &= \text{UInt}(Rt); \\
n &= \text{UInt}(Rn); \\
\text{postindex} &= \text{FALSE}; \\
\text{add} &= \text{TRUE}; \\
\text{register form} &= \text{FALSE}; \\
\text{imm32} &= \text{ZeroExtend}(\text{imm8}, 32);
\end{align*}
\]

if \( t = 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U”:

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U”:

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

\(<Rm>\) Is the general-purpose index register, encoded in the "Rm" field.

\(+\) Specifies the offset is added to the base register.

\(<\text{imm}>\) For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.
Operation

if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes as LDRH (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | != | 1111 | 0 | 0 | 0 | P | U | 1 | W | 1 | != | 1111 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| cond | Rn |
```

Offset (P == 1 && W == 0)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm}>]
```

Post-indexed (P == 0 && W == 0)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>],#{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>],#{+/-}<imm>!
```

if Rt == '1111' then SEE "LDRSB (literal)";
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12:imm4L, 32);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

```
|   | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | != | 1111 | != | 1111 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rn | Rt |
```

T1

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #<>+<imm}>]
```

if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = TRUE; add = TRUE; wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13

T2
Offset \( (Rt \neq 1111 \&\& P = 1 \&\& U = 0 \&\& W = 0) \)

\[
\text{LDRSB}\{<c>\}\{<q>\} \{<Rt>\}, \{<Rn>\} \{, \#<\text{-imm}>\}
\]

Post-indexed \( (P = 0 \&\& W = 1) \)

\[
\text{LDRSB}\{<c>\}\{<q>\} \{<Rt>\}, \{<Rn>\}, \#(+/-)<\text{imm}>
\]

Pre-indexed \( (P = 1 \&\& W = 1) \)

\[
\text{LDRSB}\{<c>\}\{<q>\} \{<Rt>\}, \{<Rn>\}, \#(+/-)<\text{imm}>\}
\]

\[
\begin{align*}
\text{if } Rt & = '1111' \&\& P = '1' \&\& U = '0' \&\& W = '0' \text{ then SEE "PLI";} \\
\text{if } Rn & = '1111' \text{ then SEE "LDRSB (literal)";} \\
\text{if } P & = '1' \&\& U = '1' \&\& W = '0' \text{ then SEE "LDRSBT";} \\
\text{if } P & = '0' \&\& W = '0' \text{ then UNDEFINED;} \\
\text{t} & = \text{UInt}(Rt); \text{ } n = \text{UInt}(Rn); \text{ } \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32); \\
\text{index} & = \text{(P == '1')} ; \text{ } \text{add} = (\text{U == '1'}); \text{ } \text{wback} = (W == '1'); \\
\text{if } (t == 15 \&\& W == '1') || (wback \&\& n == t) \text{ then UNPREDICTABLE;} \\
// \text{Armv8-A removes UNPREDICTABLE for R13}
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( wback \&\& n = t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

**Assembler Symbols**

- \(<c>\) \(\) See [Standard assembler syntax fields](#).
- \(<q>\) \(\) See [Standard assembler syntax fields](#).
- \(<Rt>\) \(\) Is the general-purpose register to be transferred, encoded in the "Rt" field.
- \(<Rn>\) \(\) Is the general-purpose base register, encoded in the "Rn" field. For PC use see [LDRSB (literal)](#).
- \(+/-\) \(\) Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

\[
\begin{array}{c|c}
U & +/-(\text{+-}) \\
\hline
0 & + \\
1 & -
\end{array}
\]

- + \(\) Specifies the offset is added to the base register.
- \(<\text{imm}>\) \(\) For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

  For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

  For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.
Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  offset_addr = if add then \( R[n] + \text{imm32} \) else \( R[n] - \text{imm32} \);
  address = if index then offset_addr else \( R[n] \);
  \( R[t] = \text{SignExtend}(\text{MemU}[\text{address},1], 32) \);
  if wback then \( R[n] = \text{offset_addr} \);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

A1 !(P == 0 & W == 1))

LDRSB{<c>}{<q>} <label> // (Normal form)
LDRSB{<c>}{<q>} <Rt>, [PC, #<imm>] // (Alternative form)

if P == '0' && W == '1' then SEE "LDRSBT";

T1

T1

LDRSB{<c>}{<q>} <Rt>, [PC, #<imm>] // (Preferred syntax)
LDRSB{<c>}{<q>} <Rt>, [PC, #<imm>] // (Alternative syntax)

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.

Rt

See Standard assembler syntax fields.

is the general-purpose register to be transferred, encoded in the "Rt" field.
For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/-. Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC, 4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address,1], 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>0</td>
<td>W</td>
<td>1</td>
<td>Rn</td>
<td>Rt</td>
<td>{0}{0}{0}{0}</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cond

Offset \( (P == 1 && W == 0) \)

LDRSB{<c>}{<q>}{<Rt>}, [<Rn>, {+/-%}<Rm>]

Post-indexed \( (P == 0 && W == 0) \)

LDRSB{<c>}{<q>}{<Rt>}, [<Rn>], {+/-%}<Rm>]

Pre-indexed \( (P == 1 && W == 1) \)

LDRSB{<c>}{<q>}{<Rt>}, [<Rn>, {+/-%}<Rm>]

\[
\begin{align*}
\text{if } P &= '0' \&\& W &= '1' \text{ then SEE "LDRSBT";} \\
t &= \text{UInt}(Rt); \quad n &= \text{UInt}(Rn); \quad m &= \text{UInt}(Rm); \\
\text{index} &= (P == '1'); \quad \text{add} &= (U == '1'); \quad wback &= (P == '0') \| (W == '1'); \\
(shift_t, shift_n) &= (\text{SRType}_{\text{LSL}}, 0); \\
\text{if } t &= 15 \| m &= 15 \text{ then UNPREDICTABLE} \\
\text{if wback } &\& (n &= 15 \| n &= t) \text{ then UNPREDICTABLE};
\end{align*}
\]

CONstrained UNPREDICTABLE behavior

If wback \( &\& n &= t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | Rm | 1 | Rn | Rt |

T1

LDRSB{<c>}{<q>}{<Rt>}, [<Rn>, {+}<Rm>]

\[
\begin{align*}
t &= \text{UInt}(Rt); \quad n &= \text{UInt}(Rn); \quad m &= \text{UInt}(Rm); \\
\text{index} &= \text{TRUE}; \quad \text{add} &= \text{TRUE}; \quad wback &= \text{FALSE}; \\
(shift_t, shift_n) &= (\text{SRType}_{\text{LSL}}, 0);
\end{align*}
\]
T2

LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th></th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSBT

Load Register Signed Byte Unprivileged loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.
The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.
LDRSBT is UNPREDICTABLE in Hyp mode.
The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.
The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 0 0 0 0 | U | 1 | 1 | Rn | Rt | imm4H | 1 | 1 | 0 | 1 | imm4L | cond |

t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If $n == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If $n == t && n != 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 0 0 0 0 | U | 0 | 1 | 1 | Rn | Rt | (0)(0)(0)(0) | 1 | 1 | 0 | 1 | Rm | cond |

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
CONSTRAINED UNPREDICTABLE behavior

If \( n == t \) && \( n \neq 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

\[
\begin{array}{ccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & !=1111 & \text{Rt} & 1 & 1 & 1 & 0 & \text{imm8} & \text{Rn}
\end{array}
\]

T1

\[
\text{LDRSBT}\{\langle c\rangle}\{\langle q\rangle\} \langle \text{Rt} \rangle, [\langle \text{Rn} \rangle \{, \#\{+\}\langle \text{imm}\rangle\}]
\]

if \( \text{Rn} == '1111' \) then SEE "LDRSB (literal)";
\[
t = \text{UInt}(\text{Rt}); \quad n = \text{UInt}(\text{Rn}); \quad \text{postindex} = \text{FALSE}; \quad \text{add} = \text{TRUE};
\]
\[
\text{register} \text{ form} = \text{FALSE}; \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32);
\]
\[
\text{if} \ t == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<\text{Rt}>\) Is the general-purpose register to be transferred, encoded in the "\text{Rt}" field.
\(<\text{Rn}>\) Is the general-purpose base register, encoded in the "\text{Rn}" field.
\(+/-\) For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

\[
\begin{array}{c|c}
\text{U} & +/- \\
0 & - \\
1 & + \\
\end{array}
\]

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

\[
\begin{array}{c|c}
\text{U} & +/- \\
0 & - \\
1 & + \\
\end{array}
\]

\(<\text{Rm}>\) Is the general-purpose index register, encoded in the "\text{Rm}" field.
\(+\) Specifies the offset is added to the base register.
\(<\text{imm}>\) For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.
Operation

if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
    if postindex then R[n] = offset_addr;

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRSB (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
| ! = 1111 |   0  0   | P | U | 1 | W | 1 | ! = 1111 |   | Rt | imm4H | 1 | 1 | 1 | 1 | imm4L |
```

cond Rn

**Offset (P == 1 & W == 0)**

LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #/\+\-}<imm>]

**Post-indexed (P == 0 & W == 0)**

LDRSH{<c>}{<q>} <Rt>, [<Rn>], #/\+\-<imm>

**Pre-indexed (P == 1 & W == 1)**

LDRSH{<c>}{<q>} <Rt>, [<Rn>], #/\+\-<imm>!

If Rn == '1111' then SEE "LDRSH (literal)";
if P == '0' & W == '1' then SEE "LDRSHT";

t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12:imm4L, 32);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
if t == 15 || (wback & n == t) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If wback & n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

```
| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0  9  8  7  6  5  4  3  2  1  0 |
| 1  1  1  1  0  0  1  1  0  1  1 | ! = 1111 | ! = 1111 | imm12 |
```

Rn Rt

T1

LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #/\+\-}<imm>]

if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";

t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = TRUE; add = TRUE; wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13

T2

LDRSH (immediate)
Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

LDRSH{<c}>{<q>} <Rt>, [<Rn>], {, #<-imm>}

Post-indexed (P == 0 && W == 1)

LDRSH{<c}>{<q>} <Rt>, [<Rn>], #<imm>

Pre-indexed (P == 1 && W == 1)

LDRSH{<c}>{<q>} <Rt>, [<Rn>], #<imm>!!

if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related instructions"
if P == '1' && U == '1' && W == '0' then SEE "LDRSHT"
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Related instructions: Load/store single.

Assembler Symbols

See Standard assembler syntax fields.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.
Operation

if \texttt{ConditionPassed()} then
  EncodingSpecificOperations();
  offset\_addr = if add then (R[n] + \texttt{imm32}) else (R[n] - \texttt{imm32});
  address = if index then offset\_addr else R[n];
  data = \texttt{MemU}[address,2];
  if \texttt{wback} then R[n] = offset\_addr;
  R[t] = \texttt{SignExtend}(data, 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| ! | 1 | 1 | 1 | 1 | 0 | 0 | 0 | P | U | 1 | W | 1 | 1 | 1 | 1 | Rt | imm4H | 1 | 1 | 1 | 1 | imm4L |
| cond |

A1 (!{P == 0 && W == 1})

LDRSH{<c>}{<q>}{<Rt>, <label>} // (Normal form)

LDRSH{<c>}{<q>}{<Rt>, [PC, #</imm>]} // (Alternative form)

if P == '0' & W == '1' then SEE "LDRSHT";

if P == '0' & W == '1' then see "LDRSHT";

t = UInt(Rt); imm32 = ZeroExtend(imm4H:imm4L, 32);

add = (U == '1'); wback = (P == '0') || (W == '1');

if t == 15 || wback then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRSH (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1

| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | U | 0 | 1 | 1 | 1 | 1 | 1 | != | 1 | 1 | 1 | 1 | imm12 | \( \text{imm12} \) |
| Rt |

T1

LDRSH{<c>}{<q>}{<Rt>, <label>} // (Preferred syntax)

LDRSH{<c>}{<q>}{<Rt>, [PC, #</imm>]} // (Alternative syntax)

if Rt == '1111' then SEE "Related instructions";

t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); add = (U == '1');

// Armv8-A removes UNPREDICTABLE for R13

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related instructions: Load, signed (literal).

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.
For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/-
Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC, 4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address, 2];
    R[t] = SignExtend(data, 32);
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111| 0  | 0  | P | U | 0 | W | 1  | Rn | Rt | (0)| (0)| (0)| (0) | 1  | 1  | 1  | 1  | Rm |
| cond

Offset (P == 1 && W == 0)

LDRSH{<c>}{<q>}{<Rt>, [<Rn>, {+/-}<Rm>]

Post-indexed (P == 0 && W == 0)

LDRSH{<c>}{<q>}{<Rt>, [<Rn>], {+/-}<Rm>]

Pre-indexed (P == 1 && W == 1)

LDRSH{<c>}{<q>}{<Rt>, [<Rn>], {+/-}<Rm>]

if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;

CONstrained unPReDiCtAbLe behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is unknown. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1

LDRSH{<c>}{<q>}{<Rt>, [<Rn>, {+}<Rm>]

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
LDRSH  \{<c>\}.W \langle<Rt>, [<Rn>, {+}\langle<Rm>\}] \} // (\langle<Rt>, <Rn>, <Rm>\}) can be represented in T1

LDRSH\{<c>\}\{<q>\} \langle<Rt>, [<Rn>, {+}\langle<Rm>\}, LSL #\langle<imm>\}]\}

if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";

If Rn == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related instructions: Load/store, signed (register offset).

Assemble Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\) For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.
For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

\[
\begin{array}{c|c}
0 & - \\
1 & + \\
\end{array}
\]

\(<Rm>\) Is the general-purpose index register, encoded in the "Rm" field.

\(<imm>\) If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

if ConditionPassed() then
EncodingSpecificOperations();
offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
offset_addr = if add then (R[n] + offset) else (R[n] - offset);
address = if index then offset_addr else R[n];
data = MemU[address,2];
if wback then R[n] = offset_addr;
R[t] = SignExtend(data, 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSHT

Load Register Signed Halfword Unprivileged loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRSHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

LDRSHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/−}<imm>}

\[
\begin{array}{cccccccccccccccccccc}
1111 & 0 & 0 & 0 & 0 & U & 1 & 1 & Rn & Rt & \text{imm4H} & 1 & 1 & 1 & 1 & \text{imm4L} \\
\end{array}
\]

cond

\[
\begin{align*}
t &= \text{UInt}(Rt); \\
n &= \text{UInt}(Rn); \\
\text{postindex} &= \text{TRUE}; \\
\text{add} &= (U == '1'); \\
\text{register\_form} &= \text{FALSE}; \\
\text{imm32} &= \text{ZeroExtend}(\text{imm4H}:\text{imm4L}, 32); \\
\text{if } t == 15 || n == 15 || n == t \text{ then UNPREDICTABLE;}
\end{align*}
\]

CONstrained UNPREDICTABLE behavior

If \( n == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If \( n == t \land n != 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2

LDRSHT{<c>}{<q>} <Rt>, [<Rn>], {+/−}<Rm>

\[
\begin{array}{cccccccccccccccccccc}
1111 & 0 & 0 & 0 & 0 & U & 0 & 1 & 1 & Rn & Rt & \text{(0)(0)(0)(0)} & 1 & 1 & 1 & 1 & \text{Rm} \\
\end{array}
\]

cond

\[
\begin{align*}
t &= \text{UInt}(Rt); \\
n &= \text{UInt}(Rn); \\
m &= \text{UInt}(Rm); \\
\text{postindex} &= \text{TRUE}; \\
\text{add} &= (U == '1'); \\
\text{register\_form} &= \text{TRUE}; \\
\text{if } t == 15 || n == 15 || n == t || m == 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]
CONSTRANDED UNPREDICTABLE behavior

If \( n == t & n != 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

LDRSHT\{<c}\}{<q>} <Rt>, \{, #\{+\}<imm>\}

if \( Rn == '1111' \) then SEE "LDRSH (literal)";
\( t = UInt(Rt) \); \( n = UInt(Rn) \); postindex = FALSE; add = TRUE;
register_form = FALSE; \( imm32 = ZeroExtend(imm8, 32) \);
if \( t == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

```
\begin{array}{c|c}
U & +/- \\
\hline
0 & - \\
1 & + \\
\end{array}
```

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

```
\begin{array}{c|c}
U & +/- \\
\hline
0 & - \\
1 & + \\
\end{array}
```

\(<Rm>\) Is the general-purpose index register, encoded in the "Rm" field.

\(+\) Specifies the offset is added to the base register.

\(<imm>\) For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.
**Operation**

if `ConditionPassed()` then
  if PSTATE.EL == `EL2` then UNPREDICTABLE; // Hyp mode
  EncodingSpecificOperations();
  offset = if register_form then R[m] else imm32;
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  address = if postindex then R[n] else offset_addr;
  data = MemU_unpriv(address, 2);
  if postindex then R[n] = offset_addr;
  R[t] = SignExtend(data, 32);

**CONSTRAINED UNPREDICTABLE behavior**

If PSTATE.EL == `EL2`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes as **LDRSH** (immediate).

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load Register Unprivileged loads a word from memory, and writes it to a register. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | \(\neq 1111\) | 0 1 0 0 | U | 0 | 1 | 1 | Rn | Rt | imm12 |
| cond |

**A1**

LDRT{<c>}{<q>} <Rt>, [<Rn>] {}, #{+/-}<imm>

t = \(\text{UInt}(Rt)\);  n = \(\text{UInt}(Rn)\);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(n == 15\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If \(n == t \&\& n != 15\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

**A2**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | \(\neq 1111\) | 0 1 1 | 0 | U | 0 | 1 | 1 | Rn | Rt | imm5 | stype | 0 | Rm |
| cond |

**A2**

LDRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

t = \(\text{UInt}(Rt)\);  n = \(\text{UInt}(Rn)\);  m = \(\text{UInt}(Rm)\);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
CONSTRAINED UNPREDICTABLE behavior

If \( n == t \) && \( n \neq 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{!} & 1 & 1 & 1 & 1
\end{array}
\]

Rt 1 1 1 0imm8

T1

\[
\text{LDRT}\{<c>}\{<q>\} \ <Rt> \ , \ <Rn> \ \{., \#\}<imm>\}
\]

if \( Rn == \text{'1111'} \) then SEE "LDR (literal)"
\[ t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \ \text{postindex} = \text{FALSE}; \ \text{add} = \text{TRUE}; \]
\[ \text{register form} = \text{FALSE}; \ \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32); \]
if \( t == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constrains on UNPREDICTABLE behaviors.

Assembler Symbols

- \(<c>\) See Standard assembler syntax fields.
- \(<q>\) See Standard assembler syntax fields.
- \(<Rt>\) For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
  
  For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.
- \(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field.
- +/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":
  \[
  \begin{array}{c|c}
  U & +/-
  \hline
  0 & - \\
  1 & +
  \end{array}
  \]
  For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":
  \[
  \begin{array}{c|c}
  U & +/-
  \hline
  0 & - \\
  1 & +
  \end{array}
  \]
- \(<Rm>\) Is the general-purpose index register, encoded in the "Rm" field.
- \(<\text{shift}>\) The shift to apply to the value read from \(<Rm>\). If absent, no shift is applied. Otherwise, see Shifts applied to a register.
- \(<+\>\) Specifies the offset is added to the base register.
- \(<\text{imm}>\) For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
  
  For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.
Operation

if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;       // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,4];
    if postindex then R[n] = offset_addr;
    R[t] = data;

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDR (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of MOV, MOVs (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVs (register).
- The description of MOV, MOVs (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

```

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | (0) | (0) | (0) | Rd | != 00000 | 0 | 0 | 0 | Rm |
```

cond S imm5 stype

MOV, shift or rotate by value

```

LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```

MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

T2

```

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 0 | 0 | 0 | 0 | != 00000 | Rm | Rd |
```

<table>
<thead>
<tr>
<th>op</th>
<th>imm5</th>
</tr>
</thead>
</table>

T2

```

LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm> // (Inside IT block)
```

is equivalent to

```

MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when InITBlock().

T3

```

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | (0) | (0) | (0) | imm3 | Rd | imm2 | 0 | 0 | Rm |
```

S stype

MOV, shift or rotate by value

```

LSL{<c>}.W (<Rd>,) <Rm>, #<imm> // (Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)
```

```

LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```

MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.
Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

<imm> For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

Not flag setting

\[
\text{LSL} \{<c>\}{<q>} \{<Rd>, <Rm>, <Rs>}
\]

is equivalent to

\[
\text{MOV} \{<c>\}{<q>} \{<Rd>, <Rm>, \text{LSL} <Rs>}
\]

and is always the preferred disassembly.

T1

Logical shift left

\[
\text{LSL}<c>{<q>} \{<Rdm>, <Rd>, <Rs> // (Inside IT block)}
\]

is equivalent to

\[
\text{MOV}<c>{<q>} \{<Rdm>, <Rd>, \text{LSL} <Rs>}
\]

and is the preferred disassembly when \text{InITBlock}().

T2

Not flag setting

\[
\text{LSL}<c>.W \{<Rd>, <Rm>, <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)}
\]

\[
\text{LSL} \{<c>\}{<q>} \{<Rd>, <Rm>, <Rs>}
\]

is equivalent to
MOV\{<c>\}{<q>} <Rd>, <Rm>, LSL <Rs>

and is always the preferred disassembly.

### Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rdm>` Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the first general-purpose source register, encoded in the "Rm" field.
- `<Rs>` Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

### Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
**LSLS (immediate)**

Logical Shift Left, setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores \textit{PSTATE} from SPSR.<current_mode>.
- The PE checks SPSR.<current_mode> for an illegal return event. See \textit{Illegal return events from AArch32 state}.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of \textit{MOV, MOVS (register)}. This means:

- The encodings in this description are named to match the encodings of \textit{MOV, MOVS (register)}.
- The description of \textit{MOV, MOVS (register)} gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

### A1

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>Rd</td>
<td>!=</td>
<td>00000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**cond** | imm5 | stype

### MOVS, shift or rotate by value

\[\text{LSLS\{<c>\}\{<q>\}\{<Rd>,\}<Rm>, \#<imm>}\]

is equivalent to

\[\text{MOVS\{<c>\}\{<q>\}\{<Rd>,\}<Rm>, \text{LSL} \#<imm>}\]

and is always the preferred disassembly.

### T2

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 0 | 0 | 0 | 0 | 0 | != | 00000 | Rm | Rd |
```

### T3

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | (0) | imm3 | Rd | imm2 | 0 | 0 | Rm |
```

S | imm3 | Rd | imm2 | 0 | 0 | Rm | stype

LSLS (immediate)
MOVS, shift or rotate by value

\[
\text{LSLS}\{<\text{Rd}>,\} <\text{Rm}>, \text{#}<\text{imm}> \quad \text{// (Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)}
\]

\[
\text{LSLS}\{<\text{c}>,\} \{<\text{q}>,\} <\text{Rm}>, \text{#}<\text{imm}>
\]

is equivalent to

\[
\text{MOVS}\{<\text{c}>,\} \{<\text{q}>,\} <\text{Rd}>, <\text{Rm}>, \text{LSL #}<\text{imm}>
\]

and is always the preferred disassembly.

Assembler Symbols

\(<\text{c}>,\) See Standard assembler syntax fields.

\(<\text{q}>,\) See Standard assembler syntax fields.

\(<\text{Rd}>,\) For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

\(<\text{Rm}>,\) For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

\(<\text{imm}>,\) For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
**LSLS (register)**

Logical Shift Left, setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of **MOV, MOVs (register-shifted register)**. This means:

- The encodings in this description are named to match the encodings of **MOV, MOVs (register-shifted register)**.
- The description of **MOV, MOVs (register-shifted register)** gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

**A1**

```
| != 1111 | 0 0 0 1 1 0 1 1 | (0)(0)(0)(0) | Rd | Rs | 0 0 0 1 | Rm |
| cond   | S             | stype        |
```

**Flag setting**

\[
\text{LSLS}\{<c>|<q>|<Rd>,}<Rm>,<Rs>
\]

is equivalent to

\[
\text{MOV}\{<c>|<q>|<Rd>,}<Rm>,\text{LSL}<Rs>
\]

and is always the preferred disassembly.

**T1**

```
| 0 1 0 0 0 0 | 0 0 1 0 | Rs | Rdm |
| op         |
```

**Logical shift left**

\[
\text{LSLS}\{<q>|<Rdm>,}<Rdm>,<Rs> // (Outside IT block)
\]

is equivalent to

\[
\text{MOV}\{<q>|<Rdm>,}<Rdm>,\text{LSL}<Rs>
\]

and is the preferred disassembly when \!InITBlock().

**T2**

```
| 1 1 1 1 1 0 | 1 0 0 | 0 0 1 | Rm | 1 1 1 1 | Rd | 0 0 0 0 | Rs |
| stype S     |
```

**Flag setting**

\[
\text{LSLS.W}\{<Rd>,}<Rm>,<Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
\]

\[
\text{LSLS}\{<c>|<q>|<Rd>,}<Rm>,<Rs>
\]

is equivalent to
MOVS\(<c>\){\(<q>\)} <Rd>, <Rm>, LSL <Rs>

and is always the preferred disassembly.

**Assembler Symbols**

\(<c>\)   See *Standard assembler syntax fields*.
\(<q>\)   See *Standard assembler syntax fields*.
\(<Rdm>\) Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
\(<Rd>\)  Is the general-purpose destination register, encoded in the "Rd" field.
\(<Rm>\)  Is the first general-purpose source register, encoded in the "Rm" field.
\(<Rs>\)  Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation**

The description of **MOV, MOVS (register-shifted register)** gives the operational pseudocode for this instruction.
LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of MOV, MOVs (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVs (register).
- The description of MOV, MOVs (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ! = | 1111 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| cond | S | stype |

MOV, shift or rotate by value

\[ \text{LSR}\{<c>\}\{<q>\} \{<Rd>,\} <Rm>, \#<imm> \]

is equivalent to

\[ \text{MOV}\{<c>\}\{<q>\} <Rd>, <Rm>, \text{LSR} \#<imm> \]

and is always the preferred disassembly.

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm5</td>
<td>Rm</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T2

\[ \text{LSR}\{<c>\}\{<q>\} \{<Rd>,\} <Rm>, \#<imm> \] // (Inside IT block)

is equivalent to

\[ \text{MOV}\{<c>\}\{<q>\} <Rd>, <Rm>, \text{LSR} \#<imm> \]

and is the preferred disassembly when InITBlock().

T3

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>stype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MOV, shift or rotate by value

\[ \text{LSR}\{<c>\}.W \{<Rd>,\} <Rm>, \#<imm> \] // (Inside IT block, and \langle Rd, Rm, \#imm \rangle can be represented in T2)

\[ \text{LSR}\{<c>\}\{<q>\} \{<Rd>,\} <Rm>, \#<imm> \]

is equivalent to

\[ \text{MOV}\{<c>\}\{<q>\} <Rd>, <Rm>, \text{LSR} \#<imm> \]

and is always the preferred disassembly.
Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

<imm> For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOV$S$ (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOV$S$ (register-shifted register).
- The description of MOV, MOV$S$ (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|-----------------|-----------------|-----------------|
| != 1111                          0 0 1 1 0 1 0 (0) (0) (0) | Rd               Rs   0 1 1 |
| cond                            S                               stype                |

Not flag setting

LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|-----------------|-----------------|-----------------|
| 0 1 0 0 0 0 0 1 1 | Rs   | Rdm |
| op                               |

Logical shift right

LSR<>{<q>} {<Rdm>,} <Rdm>, <Rs> // (Inside IT block)

is equivalent to

MOV<>{<q>} <Rdm>, <Rdm>, LSR <Rs>

and is the preferred disassembly when InITBlock().

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|-----------------|-----------------|-----------------|
| 1 1 1 1 1 0 1 0 0 1 0 | Rm   | 1 1 1 1 | Rd   | 0 0 0 0 | Rs |
| stype                            S |

Not flag setting

LSR<>.W {<Rd>,} <Rm>, <Rs> // (Inside IT block, and <Rd>, <Rm>, <Rd>, <Rs> can be represented in T1)

LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rdm>` Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the first general-purpose source register, encoded in the "Rm" field.
- `<Rs>` Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation**

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
**LSRS (immediate)**

Logical Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result. The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores \(PSTATE\) from \(SPSR_{\text{current\_mode}}\).
- The PE checks \(SPSR_{\text{current\_mode}}\) for an illegal return event. See [Illegal return events from AArch32 state](#).
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of **MOV, MOVS (register)**. This means:

- The encodings in this description are named to match the encodings of **MOV, MOVS (register)**.
- The description of **MOV, MOVS (register)** gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (\(A1\)) and T32 (\(T2\) and \(T3\)).

**A1**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| != 1111 | 0 0 0 1 1 | 0 1 1 | (0) (0) (0) (0) | Rd | imm5 | 0 1 0 | Rm |
| cond | S | stype |
```

**MOVS, shift or rotate by value**

\[
\text{LSRS}\{<c>\}\{<q>\} \{<Rd>,\} \langle Rm \rangle, \#<imm>
\]

is equivalent to

\[
\text{MOVS}\{<c>\}\{<q>\} \langle Rd \rangle, \langle Rm \rangle, \text{LSR} \#<imm>
\]

and is always the preferred disassembly.

**T2**

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 0 0 | 0 1 | imm5 | Rm | Rd |
| op |
```

**T2**

\[
\text{LSRS}\{<q>\} \{<Rd>,\} \langle Rm \rangle, \#<imm> \quad \text{// (Outside IT block)}
\]

is equivalent to

\[
\text{MOVS}\{<q>\} \langle Rd \rangle, \langle Rm \rangle, \text{LSR} \#<imm>
\]

and is the preferred disassembly when \(!\text{InITBlock}()\).

**T3**

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 1 1 | 0 1 0 0 | 1 0 0 | imm3 | Rd | imm2 | 0 1 | Rm |
| S | stype |
```
MOVS, shift or rotate by value

LSRS.W {<Rd>,} <Rm>, #<imm> // (Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)

LSRS{<c>{<q>}} {<Rd>,} <Rm>, #<imm>

is equivalent to

MOVS{<c>{<q>}} <Rd>, <Rm>, LSR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

<imm> For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
LSRS (register)

Logical Shift Right, setting flags (register) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

```
| ! = 1111 | 0 0 0 1 1 0 1 1 (0) (0) (0) | Rd | Rs | 0 0 1 1 | Rm |
| cond | S | stype |
```

Flag setting

LSRS{<c}>{<q>} {<Rd>}, <Rm>, <Rs>

is equivalent to

MOVS{<c}>{<q>} <Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

T1

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

Logical shift right

LSRS{<q>} {<Rdm>}, <Rd>, <Rs> // (Outside IT block)

is equivalent to

MOVS{<q>} <Rdm>, <Rd>, LSR <Rs>

and is the preferred disassembly when !InITBlock().

T2

```
<table>
<thead>
<tr>
<th>stype</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 0 0</td>
<td>Rm</td>
</tr>
</tbody>
</table>
```

Flag setting

LSRS.W {<Rd>}, <Rm>, <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)

LSRS{<c}>{<q>} {<Rd>}, <Rm>, <Rs>

is equivalent to
MOVS<{c}>{<q>} <Rd>, <Rm>, LSR <Rs>
and is always the preferred disassembly.

Assembler Symbols

<c>   See Standard assembler syntax fields.

<q>   See Standard assembler syntax fields.

<Rdm> Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

<Rd>  Is the general-purpose destination register, encoded in the "Rd" field.

<Rm>  Is the first general-purpose source register, encoded in the "Rm" field.

<Rs>  Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
MCR

Move to System register from general-purpose register or execute a System instruction. This instruction copies the value of a general-purpose register to a System register, or executes a System instruction. The System register and System instruction descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface and General behavior of System registers.

In an implementation that includes EL2, MCR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable instruction enables, disables, and traps.

Because of the range of possible traps to Hyp mode, the MCR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 1111</td>
</tr>
</tbody>
</table>

cond coproc<3:1>

A1

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

t = UInt(Rt); cp = if coproc<0> == '0' then 14 else 15;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
</tr>
</tbody>
</table>

coproc<3:1>

T1

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

t = UInt(Rt); cp = if coproc<0> == '0' then 14 else 15;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<coproc> Is the System register encoding space, encoded in “coproc<0>”:

<table>
<thead>
<tr>
<th>coproc&lt;0&gt;</th>
<th>&lt;coproc&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p14</td>
</tr>
<tr>
<td>1</td>
<td>p15</td>
</tr>
</tbody>
</table>

<opc1> Is the opc1 parameter within the System register encoding space, in the range 0 to 7, encoded in the “opc1” field.

<Rt> Is the general-purpose register to be transferred, encoded in the “Rt” field.

<CRn> Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRn” field.
<CRm> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.
<opc2> Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc2" field.

The possible values of { <coproc>, <opc1>, <CRn>, <CRm>, <opc2> } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

**Operation**

if `ConditionPassed()` then
  EncodingSpecificOperations();
  `AArch32.SysRegWrite(cp, ThisInstr(), R[t]);`

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MCRR

Move to System register from two general-purpose registers. This instruction copies the values of two general-purpose registers to a System register.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface and General behavior of System registers.

In an implementation that includes EL2, MCRR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCRR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable instruction enables, disables, and traps.

Because of the range of possible traps to Hyp mode, the MCRR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
<tr>
<td>t = UInt(Rt); t2 = UInt(Rt2); cp = if coproc&lt;0&gt; == '0' then 14 else 15;</td>
</tr>
<tr>
<td>if t == 15</td>
</tr>
<tr>
<td>// Armv8-A removes UNPREDICTABLE for R13</td>
</tr>
</tbody>
</table>

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1 0 0</td>
</tr>
<tr>
<td>coproc&lt;3:1&gt;</td>
</tr>
<tr>
<td>t = UInt(Rt); t2 = UInt(Rt2); cp = if coproc&lt;0&gt; == '0' then 14 else 15;</td>
</tr>
<tr>
<td>if t == 15</td>
</tr>
<tr>
<td>// Armv8-A removes UNPREDICTABLE for R13</td>
</tr>
</tbody>
</table>

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<coproc> Is the System register encoding space, encoded in "coproc<0>":

<table>
<thead>
<tr>
<th>coproc&lt;0&gt;</th>
<th>&lt;coproc&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p14</td>
</tr>
<tr>
<td>1</td>
<td>p15</td>
</tr>
</tbody>
</table>

<opc1> Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.

<Rt> Is the first general-purpose register that is transferred into, encoded in the "Rt" field.

<Rt2> Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.
<CRm> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of { <coproc>, <opc1>, <CRm> } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <Rt2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    value = R[t2]:R[t];
    AArch32.SysRegWrite64(cp, ThisInstr(), value);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MLA, MLAS

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

In an A32 instruction, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | S | Rd | Ra | Rm | 1 | 0 | 0 | 1 | Rn |
```

Flag setting (S == 1)

MLAS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Not flag setting (S == 0)

MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); setflags = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;

T1

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | Rn | != | 1111 | Rd | 0 | 0 | 0 | 0 | Rm |
```

T1

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | Ra | != | 1111 | Rd | 0 | 0 | 0 | 0 | Rm |
```

if Ra == '1111' then SEE "MUL";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); setflags = FALSE;
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Ra>` Is the third general-purpose source register holding the addend, encoded in the "Ra" field.
Operation

if `ConditionPassed()` then
    EncodingSpecificOperations();
    operand1 = `SInt(R[n])`;  // operand1 = `UInt(R[n])` produces the same final results
    operand2 = `SInt(R[m])`;  // operand2 = `UInt(R[m])` produces the same final results
    addend = `SInt(R[a])`;  // addend = `UInt(R[a])` produces the same final results
    result = operand1 * operand2 + addend;
    `R[d]` = result<31:0>;
    if `setflags` then
        PSTATE.N = result<31>;
        PSTATE.Z = `IsZeroBit(result<31:0>)`;
        // PSTATE.C, PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Multiply and Subtract multiplies two register values, and subtracts the product from a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | | Rd | Ra | Rm | | | | | | | | | | | | | | | | | 1 | 0 | 0 | 1 | Rn |
```

```
cond
```

**T1**

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | | Rn | Ra | Rd | 0 | 0 | 0 | 1 | Rm |
```

```
MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

// Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Ra>` Is the third general-purpose source register holding the minuend, encoded in the "Ra" field.

**Operation**

```java
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend = SInt(R[a]);    // addend = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Move (immediate) writes an immediate value to the destination register. If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The MOV variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1, T2 and T3).

A1

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

| != 1111 | 0 0 1 1 1 0 1 S |(0)(0)(0)| Rd | imm12 |

cond

MOV (S == 0)

MOV{<c>}{<q>} <Rd>, #<const>

MOVS (S == 1)

MOVS{<c>}{<q>} <Rd>, #<const>

\[ d = \text{UInt}(\text{Rd}); \text{ setflags} = (S == '1'); (\text{imm32}, \text{carry}) = \text{A32ExpandImm}_C(\text{imm12}, \text{PSTATE}.C); \]

A2

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

| != 1111 | 0 0 1 1 0 0 |0 0 | imm4 | Rd | imm12 |

cond

A2

MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can not be represented in A1)

MOVW{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in A1)

\[ d = \text{UInt}(\text{Rd}); \text{ setflags} = \text{FALSE}; \text{ imm32} = \text{ZeroExtend}(\text{imm4:imm12}, 32); \]

if d == 15 then UNPREDICTABLE;

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 0 1 |0 0 | Rd | imm8 |
**T1**

MOV<c>{<q>} <Rd>, #<imm8> // (Inside IT block)

MOVS{<q>} <Rd>, #<imm8> // (Outside IT block)

d = UInt(Rd); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32); carry = PSTATE.C;

**T2**

```
1 1 1 1 1 0 | i 0 0 0 1 0 5 | 1 1 1 1 0 | imm3 | Rd | imm8
```

MOV (S == 0)

MOV<c>.W <Rd>, #<const> // (Inside IT block, and <Rd>, <const> can be represented in T1)

MOV{<c>}{<q>} <Rd>, #<const>

MOVS (S == 1)

MOVS.W <Rd>, #<const> // (Outside IT block, and <Rd>, <const> can be represented in T1)

```
1 1 1 1 1 0 | i 1 0 0 1 0 0 | imm4 | 0 | imm3 | Rd | imm8
```

**T3**

```
1 1 1 1 1 0 | i 1 0 0 1 0 0 | imm4 | 0 | imm3 | Rd | imm8
```

MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> cannot be represented in T1 or T2)

MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in T1 or T2)

d = UInt(Rd); setflags = (S == '1'); (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>`
  - See Standard assembler syntax fields.
- `<q>`
  - See Standard assembler syntax fields.
- `<Rd>`
  - For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:
    - For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
    - For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
  - For encoding A2, T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
- `<imm8>`
  - Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
- `<imm16>`
  - For encoding A2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.
For encoding T3: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:imm3:imm8” field.

For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T2: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

\[
\text{if } \text{ConditionPassed}() \text{ then}
\]
\[
\text{EncodingSpecificOperations();}
\]
\[
\text{result} = \text{imm32;}
\]
\[
\text{if } d == 15 \text{ then } // \text{ Can only occur for encoding A1}
\]
\[
\text{if setflags then}
\]
\[
\text{ALUExceptionReturn(result);}
\]
\[
\text{else}
\]
\[
\text{ALUWritePC(result);}
\]
\[
\text{else}
\]
\[
R[d] = \text{result;}
\]
\[
\text{if setflags then}
\]
\[
PSTATE.N = \text{result<31>};
\]
\[
PSTATE.Z = \text{IsZeroBit(result);}
\]
\[
PSTATE.C = \text{carry;}
\]
\[
\text{// PSTATE.V unchanged}
\]

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Move (register) copies a value from a register to the destination register.

If the destination register is not the PC, the MOV variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The MOV variant of the instruction is a branch. In the T32 instruction set (encoding T1) this is a simple branch, and in the A32 instruction set it is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The MOV variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is used by the aliases ASRS (immediate), ASR (immediate), LSLS (immediate), LSL (immediate), LSRS (immediate), LSR (immediate), RORS (immediate), ROR (immediate), RRXS, and RRX.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | S | (0) | (0) | (0) | Rd | imm5 | stype | 0 | Rm |

MOV, rotate right with extend (S == 0 & imm5 == 00000 & stype == 11)

MOV{<c>}{<q>} <Rd>, <Rm>, RRX

MOV, shift or rotate by value (S == 0 & !(imm5 == 00000 & stype == 11))

MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

MOV, rotate right with extend (S == 1 & imm5 == 00000 & stype == 11)

MOV{<c>}{<q>} <Rd>, <Rm>, RRX

MOV, shift or rotate by value (S == 1 & !(imm5 == 00000 & stype == 11))

MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | D | Rm | Rd |

T1

MOV{<c>}{<q>} <Rd>, <Rm>

d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 & InITBlock() & !LastInITBlock() then UNPREDICTABLE;
T2

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 != 11 imm5 Rm Rd
op

T2

MOV{<q>} <Rd>, <Rm> {<shift> #<amount>} // (Inside IT block)
MOV{<q>} <Rd>, <Rm> {<shift> #<amount>} // (Outside IT block)

\[ d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{setflags} = \text{InITBlock}(); \]
\[(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{op}, \text{imm5}); \]
if op == '00' && imm5 == '00000' && InITBlock() then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If op == '00' && imm5 == '00000' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passed its condition code check.
- The instruction executes as NOP, as if it failed its condition code check.
- The instruction executes as MOV Rd, Rm.

T3

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 0 0 0 1 1 1 0 | imm3 | Rd | imm2 stype | Rm

MOV, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

MOV{<c>}{<q>} <Rd>, <Rm>, RRX

MOV, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

MOV{<c>}.W <Rd>, <Rm> {<shift> #<amount>} // (Inside IT block, and <Rd>, <Rm>, <shift>, <amount> can be represented in T1)
MOV{<c>}.W <Rd>, <Rm> {<shift> #<amount>} // (Outside IT block, and <Rd>, <Rm>, <shift>, <amount> can be represented in T1 or T2)

MOV{<c>}{<q>} <Rd>, <Rm> {<shift> #<amount>}

MOVS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

MOVS{<c>}{<q>} <Rd>, <Rm>, RRX

MOVS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

MOVS.W <Rd>, <Rm> {<shift> #<amount>} // (Outside IT block, and <Rd>, <Rm>, <shift>, <amount> can be represented in T1 or T2)
MOVS{<c>}{<q>} <Rd>, <Rm> {<shift> #<amount>}

\[ d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S == '1'); \]
\[(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm3:imm2}); \]
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
See *Standard assembler syntax fields*.

### <Rd>
For encoding A1: the general-purpose destination register, encoded in the "Rd" field. If the PC is used:
- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. Arm deprecates use of the instruction if <Rn> is the PC.
- For the MOVS variant, the instruction performs an exception return, that restores `PSTATE` from SPSR <current mode>. Arm deprecates use of the instruction if <Rn> is not the LR, or if the optional shift or RRX argument is specified.

For encoding T1: the general-purpose destination register, encoded in the "D:Rd" field. If the PC is used:
- The instruction causes a branch to the address moved to the PC. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The instruction must either be outside an IT block or the last instruction of an IT block.

For encoding T2 and T3: the general-purpose destination register, encoded in the "Rd" field.

### <Rm>
For encoding A1 and T1: the general-purpose source register, encoded in the "Rm" field. The PC can be used. Arm deprecates use of the instruction if <Rd> is the PC.

For encoding T2 and T3: the general-purpose source register, encoded in the "Rm" field.

### <shift>
For encoding A1 and T3: the type of shift to be applied to the source register, encoded in "stype":

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

For encoding T2: is the type of shift to be applied to the source register, encoded in "op":

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

### <amount>
For encoding A1: is the shift amount, in the range 0 to 31 (when <shift> = LSL), or 1 to 31 (when <shift> = ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 0 to 31 (when <shift> = LSL) or 1 to 31 (when <shift> = ROR), or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASRS (immediate)</td>
<td>T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)</td>
<td>$ == '1' &amp;&amp; stype == '10'</td>
</tr>
<tr>
<td>ASRS (immediate)</td>
<td>T2</td>
<td>op == '10' &amp;&amp; !InITBlock()</td>
</tr>
<tr>
<td>ASR (immediate)</td>
<td>T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)</td>
<td>$ == '0' &amp;&amp; stype == '10'</td>
</tr>
<tr>
<td>ASR (immediate)</td>
<td>T2</td>
<td>op == '10' &amp;&amp; InITBlock()</td>
</tr>
<tr>
<td>LSLS (immediate)</td>
<td>T3 (MOVS, shift or rotate by value)</td>
<td>$ == '1' &amp;&amp; imm3:Rd:imm2 != '00000000' &amp;&amp; stype == '00'</td>
</tr>
<tr>
<td>Alias</td>
<td>Of variant</td>
<td>Is preferred when</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>LSLS</strong></td>
<td>A1 (MOV, shift or rotate by value)</td>
<td>$S == '1' &amp;&amp; \text{imm5} != '00000' &amp;&amp; \text{stype} == '00'$</td>
</tr>
<tr>
<td><strong>LSLS</strong></td>
<td>T2</td>
<td>op == '00' &amp;&amp; \text{imm5} != '00000' &amp;&amp; \text{!InITBlock()}</td>
</tr>
<tr>
<td><strong>LSL</strong></td>
<td>T3 (MOV, shift or rotate by value)</td>
<td>$S == '0' &amp;&amp; \text{imm3:Rd:imm2} != '000xxxx00' &amp;&amp; \text{stype} == '00'$</td>
</tr>
<tr>
<td><strong>LSL</strong></td>
<td>A1 (MOV, shift or rotate by value)</td>
<td>$S == '0' &amp;&amp; \text{imm5} != '00000' &amp;&amp; \text{stype} == '00'$</td>
</tr>
<tr>
<td><strong>LSL</strong></td>
<td>T2</td>
<td>op == '00' &amp;&amp; \text{imm5} != '00000' &amp;&amp; \text{InITBlock()}</td>
</tr>
<tr>
<td><strong>LSRS</strong></td>
<td>T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)</td>
<td>$S == '1' &amp;&amp; \text{stype} == '01'$</td>
</tr>
<tr>
<td><strong>LSRS</strong></td>
<td>T2</td>
<td>op == '01' &amp;&amp; \text{!InITBlock()}</td>
</tr>
<tr>
<td><strong>LSR</strong></td>
<td>T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)</td>
<td>$S == '0' &amp;&amp; \text{stype} == '01'$</td>
</tr>
<tr>
<td><strong>LSR</strong></td>
<td>T2</td>
<td>op == '01' &amp;&amp; \text{InITBlock()}</td>
</tr>
<tr>
<td><strong>RORS</strong></td>
<td>T3 (MOV, shift or rotate by value)</td>
<td>$S == '1' &amp;&amp; \text{imm3:Rd:imm2} != '000xxxx00' &amp;&amp; \text{stype} == '11'$</td>
</tr>
<tr>
<td><strong>RORS</strong></td>
<td>A1 (MOV, shift or rotate by value)</td>
<td>$S == '1' &amp;&amp; \text{imm5} != '00000' &amp;&amp; \text{stype} == '11'$</td>
</tr>
<tr>
<td><strong>ROR</strong></td>
<td>T3 (MOV, shift or rotate by value)</td>
<td>$S == '0' &amp;&amp; \text{imm3:Rd:imm2} != '000xxxx00' &amp;&amp; \text{stype} == '11'$</td>
</tr>
<tr>
<td><strong>ROR</strong></td>
<td>A1 (MOV, shift or rotate by value)</td>
<td>$S == '0' &amp;&amp; \text{imm5} != '00000' &amp;&amp; \text{stype} == '11'$</td>
</tr>
<tr>
<td><strong>RRXS</strong></td>
<td>T3 (MOV, rotate right with extend)</td>
<td>$S == '1' &amp;&amp; \text{imm3} == '000' &amp;&amp; \text{imm2} == '00' &amp;&amp; \text{stype} == '11'$</td>
</tr>
<tr>
<td><strong>RRXS</strong></td>
<td>A1 (MOV, rotate right with extend)</td>
<td>$S == '1' &amp;&amp; \text{imm5} == '00000' &amp;&amp; \text{stype} == '11'$</td>
</tr>
<tr>
<td><strong>RRX</strong></td>
<td>T3 (MOV, rotate right with extend)</td>
<td>$S == '0' &amp;&amp; \text{imm3} == '000' &amp;&amp; \text{imm2} == '00' &amp;&amp; \text{stype} == '11'$</td>
</tr>
<tr>
<td><strong>RRX</strong></td>
<td>A1 (MOV, rotate right with extend)</td>
<td>$S == '0' &amp;&amp; \text{imm5} == '00000' &amp;&amp; \text{stype} == '11'$</td>
</tr>
</tbody>
</table>
Operation

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = shifted;
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases ASRS (register), ASR (register), LSLS (register), LSL (register), LSRS (register), LSR (register), RORS (register), and ROR (register).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

**Flag setting (S == 1)**

MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

**Not flag setting (S == 0)**

MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 0 0 x x x</td>
</tr>
</tbody>
</table>

op
Arithmetic shift right (op == 0100)

MOV<c>{<q>}, <Rdm>, ASR <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs> // (Outside IT block)

Logical shift left (op == 0010)

MOV<c>{<q>}, <Rdm>, LSL <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs> // (Outside IT block)

Logical shift right (op == 0011)

MOV<c>{<q>}, <Rdm>, LSR <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs> // (Outside IT block)

Rotate right (op == 0111)

MOV<c>{<q>}, <Rdm>, ROR <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs> // (Outside IT block)

if !(op IN {'0010', '0011', '0100', '0111'}) then SEE "Related encodings";

  d = UInt(Rdm);  m = UInt(Rdm);  s = UInt(Rs);
  setflags = !InITBlock();  shift_t = DecodeRegShift(op<2>:op<0>);

T2

| 1 1 1 1 1 1 1 1 1 0 1 0 0 | stype | S | Rm | 1 1 1 1 | Rd | 0 0 0 0 | Rs |

Flag setting (S == 1)

MOV{.W} <Rd>, <Rm>, <shift> <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)

MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

Not flag setting (S == 0)

MOV{.W} <Rd>, <Rm>, <shift> <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)

MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

if d == 15 || m == 15 || s == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Related encodings: In encoding T1, for an op field value that is not described above, see Data-processing (two low registers).

For more information about the CONstrained UNPredictable behavior of this instruction, see Architectural Constraints on UNPredictable behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rdm> Is the general-purpose source register and the destination register, encoded in the "Rdm" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> Is the general-purpose source register, encoded in the "Rm" field.
Is the type of shift to be applied to the second source register, encoded in “stype”:

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

Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASRS (register)</td>
<td>A1 (flag setting)</td>
<td>$S == '1' &amp;&amp; stype == '10'$</td>
</tr>
<tr>
<td>ASRS (register)</td>
<td>T1 (arithmetic shift right)</td>
<td>$op == '0100' &amp;&amp; !\text{InITBlock}()$</td>
</tr>
<tr>
<td>ASRS (register)</td>
<td>T2 (flag setting)</td>
<td>stype == '10' &amp;&amp; $S == '1'$</td>
</tr>
<tr>
<td>ASR (register)</td>
<td>A1 (not flag setting)</td>
<td>$S == '0' &amp;&amp; stype == '10'$</td>
</tr>
<tr>
<td>ASR (register)</td>
<td>T1 (arithmetic shift right)</td>
<td>$op == '0100$ &amp;&amp; \text{InITBlock}()$</td>
</tr>
<tr>
<td>ASR (register)</td>
<td>T2 (not flag setting)</td>
<td>stype == '10' &amp;&amp; $S == '0'$</td>
</tr>
<tr>
<td>LSLS (register)</td>
<td>A1 (flag setting)</td>
<td>$S == '1' &amp;&amp; stype == '00'$</td>
</tr>
<tr>
<td>LSLS (register)</td>
<td>T1 (logical shift left)</td>
<td>$op == '0010' &amp;&amp; !\text{InITBlock}()$</td>
</tr>
<tr>
<td>LSLS (register)</td>
<td>T2 (flag setting)</td>
<td>stype == '00' &amp;&amp; $S == '1'$</td>
</tr>
<tr>
<td>LSL (register)</td>
<td>A1 (not flag setting)</td>
<td>$S == '0' &amp;&amp; stype == '00'$</td>
</tr>
<tr>
<td>LSL (register)</td>
<td>T1 (logical shift left)</td>
<td>$op == '0010' &amp;&amp; !\text{InITBlock}()$</td>
</tr>
<tr>
<td>LSL (register)</td>
<td>T2 (not flag setting)</td>
<td>stype == '00' &amp;&amp; $S == '1'$</td>
</tr>
<tr>
<td>LSRS (register)</td>
<td>A1 (flag setting)</td>
<td>$S == '1' &amp;&amp; stype == '01'$</td>
</tr>
<tr>
<td>LSRS (register)</td>
<td>T1 (logical shift right)</td>
<td>$op == '0011' &amp;&amp; !\text{InITBlock}()$</td>
</tr>
<tr>
<td>LSR (register)</td>
<td>T2 (flag setting)</td>
<td>stype == '01' &amp;&amp; $S == '1'$</td>
</tr>
<tr>
<td>LSR (register)</td>
<td>A1 (not flag setting)</td>
<td>$S == '0' &amp;&amp; stype == '01'$</td>
</tr>
<tr>
<td>LSR (register)</td>
<td>T1 (logical shift right)</td>
<td>$op == '0011' &amp;&amp; !\text{InITBlock}()$</td>
</tr>
<tr>
<td>LSR (register)</td>
<td>T2 (not flag setting)</td>
<td>stype == '01' &amp;&amp; $S == '0'$</td>
</tr>
<tr>
<td>RORS (register)</td>
<td>A1 (flag setting)</td>
<td>$S == '1' &amp;&amp; stype == '11'$</td>
</tr>
<tr>
<td>RORS (register)</td>
<td>T1 (rotate right)</td>
<td>$op == '0111' &amp;&amp; !\text{InITBlock}()$</td>
</tr>
<tr>
<td>ROR (register)</td>
<td>T2 (flag setting)</td>
<td>stype == '11' &amp;&amp; $S == '1'$</td>
</tr>
<tr>
<td>ROR (register)</td>
<td>A1 (not flag setting)</td>
<td>$S == '0' &amp;&amp; stype == '11'$</td>
</tr>
<tr>
<td>ROR (register)</td>
<td>T1 (rotate right)</td>
<td>$op == '0111' &amp;&amp; !\text{InITBlock}()$</td>
</tr>
<tr>
<td>ROR (register)</td>
<td>T2 (not flag setting)</td>
<td>stype == '11' &amp;&amp; $S == '0'$</td>
</tr>
</tbody>
</table>

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (result, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | imm4 | Rd | | | | | | | | | | | | | | | | | | | |

**cond**

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>i</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>imm4</td>
<td>0</td>
<td>imm3</td>
<td>Rd</td>
</tr>
</tbody>
</table>

**A1**

MOVT{<c>}{<q>} <Rd>, #<imm16>

d = UInt(Rd); imm16 = imm4:imm12;
if d == 15 then UNPREDICTABLE;

**T1**

MOVT{<c>}{<q>} <Rd>, #<imm16>

d = UInt(Rd); imm16 = imm4:i:imm3:imm8;
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<imm16>` For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.
  
  For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

### Operation

if `ConditionPassed()` then
  EncodingSpecificOperations();
  R[d]<31:16> = imm16;
  // R[d]<15:0> unchanged

### Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Move to general-purpose register from System register. This instruction copies the value of a System register to a general-purpose register.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface and General behavior of System registers.

In an implementation that includes EL2, MRC accesses to system control registers can be trapped to Hyp mode, meaning that an attempt to execute an MRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable instruction enables, disables, and traps.

Because of the range of possible traps to Hyp mode, the MRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
   ! = 1111 | 1 | 1 | 1 | 0 | opc1 | 1 | CRn | 1 | 1 | 1 | coproc<0> | opc2 | 1 | CRm
   cond  coproc<3:1>
```

A1

```
MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}
```

```
t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

T1

```
   0 1 1 1 0 1 1 1 1 | opc1 | 1 | CRn | 1 | 1 | 1 | coproc<0> | opc2 | 1 | CRm
   coproc<3:1>
```

T1

```
MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}
```

```
t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<coproc>` Is the System register encoding space, encoded in “coproc<0>”:
  ```
  coproc<0> | <coproc>
  0  | p14
  1  | p15
  ```
- `<opc1>` Is the opc1 parameter within the System register encoding space, in the range 0 to7, encoded in the "opc1" field.
- `<Rt>` Is the general-purpose register to be transferred or APSR_nzcv (encoded as 0b1111), encoded in the "Rt" field. If APSR_nzcv is used, bits [31:28] of the transferred value are written to the PSTATE condition flags.
- `<CRn>` Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRn" field.
- `<CRm>` Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.
<opc2> Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc2" field.

The possible values of \{ <coproc>, <opc1>, <CRn>, <CRm>, <opc2> \} encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) value = AArch32.SysRegRead(cp, ThisInstr());
    if t ! = 15 then
        R[t] = value;
    elsif AArch32.SysRegReadCanWriteAPSR(cp, ThisInstr()) then
        PSTATE.<N,Z,C,V> = value<31:28>;
        // value<27:0> are not used.
    else
        UNPREDICTABLE;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MRRC

Move to two general-purpose registers from System register. This instruction copies the value of a System register to
two general-purpose registers.
The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For
more information see About the AArch32 System register interface and General behavior of System registers.
In an implementation that includes EL2, MRRC accesses to System registers can be trapped to Hyp mode, meaning that
an attempt to execute an MRRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in
the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable
instruction enables, disables, and traps.
Because of the range of possible traps to Hyp mode, the MRRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|---------------------------------------------|
| != 1111 | 1 1 0 0 | 0 | 1 | 0 | 1 | Rt2 | Rt | 1 1 1 | coproc<0> | opc1 | CRm|

cond coproc<3:1>

A1

MRRC{<c>{<q>}} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

t = UInt(Rt); t2 = UInt(Rt2); cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|---------------------------------------------|
| 1 1 1 0 1 1 0 0 | 0 | 1 | 0 | 1 | Rt2 | Rt | 1 1 1 | coproc<0> | opc1 | CRm|

coproc<3:1>

T1

MRRC{<c>{<q>}} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

t = UInt(Rt); t2 = UInt(Rt2); cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints
on UNPREDICTABLE behaviors.
Assembler Symbols

<coproc>
Is the System register encoding space, encoded in "coproc<0>":

<table>
<thead>
<tr>
<th>coproc&lt;0&gt;</th>
<th>&lt;coproc&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p14</td>
</tr>
<tr>
<td>1</td>
<td>p15</td>
</tr>
</tbody>
</table>

<opc1>
Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.

<Rt>
Is the first general-purpose register that is transferred into, encoded in the "Rt" field.

<Rt2>
Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.

<CRm>
Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of {<coproc>, <opc1>, <CRm> } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <Rt2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    value = AArch32.SysRegRead64(cp, ThisInstr());
    R[t] = value<31:0>;
    R[t2] = value<63:32>;

MRS

Move Special register to general-purpose register moves the value of the APSR, CPSR, or SPSR.<current_mode> into a general-purpose register.

Arm recommends the APSR form when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see APSR.

An MRS that accesses the SPSRs is UNPREDICTABLE if executed in User mode or System mode.

An MRS that is executed in User mode and accesses the CPSR returns an UNKNOWN value for the CPSR. {E, A, I, F, M} fields.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
! = 1111 0 0 0 1 0 | R | 0 | 0 | (1)(1)(1)(1) | Rd | (0)|(0) | 0 | (0) | 0 | 0 | 0 | (0)(0)(0)(0)
cond
```

A1

```
MRS{<c}<{q}> <Rd>, <spec_reg>

d = UInt(Rd); read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 1 1 1 1 | R | (1)(1)(1)(1) | 1 | 0 | (0) | 0 | Rd | (0)|(0) | 0 | (0)(0)(0)(0)(0)
```

T1

```
MRS{<c}<{q}> <Rd>, <spec_reg>

d = UInt(Rd); read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<spec_reg> Is the special register to be accessed, encoded in “R”:

<table>
<thead>
<tr>
<th>R</th>
<th>&lt;spec_reg&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPSR</td>
</tr>
<tr>
<td>1</td>
<td>SPSR</td>
</tr>
</tbody>
</table>

MRS
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
if read_spsr then
    if PSTATE.M IN {M32_User, M32_System} then
        UNPREDICTABLE;
    else
        R[d] = SPSR[];
else
    // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
    bits(32) mask = '11110000 00001111 00000011 11011111';
    if HavePANExt() then
        mask<22> = '1';
    if HaveDITExt() then
        mask<21> = '1';
    psr_val = GetPSRFromPSTATE(AArch32_NonDebugState) AND mask;
    if PSTATE.EL == EL0 then
        // If accessed from User mode return UNKNOWN values for E, A, I, F bits, bits<9:6>,
        // and for the M field, bits<4:0>
        psr_val<22> = bits(1) UNKNOWN;
        psr_val<9:6> = bits(4) UNKNOWN;
        psr_val<4:0> = bits(5) UNKNOWN;
    R[d] = psr_val;

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User, M32_System} && read_spsr, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
MRS (Banked register)

Move to Register from Banked or Special register moves the value from the Banked general-purpose register or Saved Program Status Registers (SPSRs) of the specified mode, or the value of ELR_hyp, to a general-purpose register. MRS (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MRS (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MRS (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see Usage restrictions on the Banked register transfer instructions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | 1 | 0 | R | 0 | 0 | M1 | Rd | (0) | (0) | 1 | M | 0 | 0 | 0 | 0 | (0) | (0) | (0) | (0) |
```

cond

A1

```
MRS{<c>}{<q>}, <banked_reg>

d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
SYSm = M:M1;
```

T1

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | R | M1 | 1 | 0 | (0) | 0 | Rd | (0) | (0) | 1 | M | (0) | (0) | (0) | (0) |
```

T1

```
MRS{<c>}{<q>}, <banked_reg>

d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
SYSm = M:M1;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

```
<cond>  See Standard assembler syntax fields.
<q>    See Standard assembler syntax fields.
<Rd>   Is the general-purpose destination register, encoded in the "Rd" field.
<banked_reg>  Is the name of the banked register to be transferred to or from, encoded in “R:M:M1”:
```
<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
<th>\text{&lt;banked_reg&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>R8_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>R9_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0010</td>
<td>R10_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0011</td>
<td>R11_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0100</td>
<td>R12_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0101</td>
<td>SP_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0110</td>
<td>LR_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0111</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>R8_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1001</td>
<td>R9_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1010</td>
<td>R10_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1011</td>
<td>R11_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1100</td>
<td>R12_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>SP_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1110</td>
<td>LR_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1111</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0000</td>
<td>LR_irq</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0001</td>
<td>SP_irq</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0010</td>
<td>LR_svc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0011</td>
<td>SP_svc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0100</td>
<td>LR_abt</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0101</td>
<td>SP_abt</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0110</td>
<td>LR_und</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0111</td>
<td>SP_und</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10xx</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1100</td>
<td>LR_mon</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1101</td>
<td>SP_mon</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1110</td>
<td>ELR_hyp</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>SP_hyp</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0xxx</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10xx</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110x</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1110</td>
<td>SPSR_fiq</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1111</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0000</td>
<td>SPSR_irq</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0010</td>
<td>SPSR_svc</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0011</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0100</td>
<td>SPSR_abt</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0101</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0110</td>
<td>SPSR_und</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0111</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10xx</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1100</td>
<td>SPSR_mon</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1101</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1110</td>
<td>SPSR_hyp</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>UNPREDICTABLE</td>
</tr>
</tbody>
</table>
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
if PSTATE.EL == EL0 then
    UNPREDICTABLE;
else
    mode = PSTATE.M;
    if read_spsr then
        SPSRaccessValid(SYSm, mode);  // Check for UNPREDICTABLE cases
        case SYSm of
            when '01110'  \[d\] = SPSR_fiq<31:0>;
            when '10000'  \[d\] = SPSR_irq<31:0>;
            when '10010'  \[d\] = SPSR_svc<31:0>;
            when '10100'  \[d\] = SPSR_abt<31:0>;
            when '10110'  \[d\] = SPSR_und<31:0>;
            when '11100'
                if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                \[d\] = SPSR_mon;
            when '11110'  \[d\] = SPSR_hyp<31:0>;
        else
            BankedRegisterAccessValid(SYSm, mode);  // Check for UNPREDICTABLE cases
            case SYSm of
                when '00xxx'  \[d\] = Rmode(m, M32_User);
                when '01xxx'  \[d\] = Rmode(m, M32_FIQ);
                when '1000x'
                    m = 14 - UInt(SYSm<0>);  // LR when SYSm<0> == 0, otherwise SP
                    \[d\] = Rmode(m, M32_IRQ);
                when '1001x'
                    m = 14 - UInt(SYSm<0>);  // LR when SYSm<0> == 0, otherwise SP
                    \[d\] = Rmode(m, M32_Svc);
                when '1010x'
                    m = 14 - UInt(SYSm<0>);  // LR when SYSm<0> == 0, otherwise SP
                    \[d\] = Rmode(m, M32_Abort);
                when '1011x'
                    m = 14 - UInt(SYSm<0>);  // LR when SYSm<0> == 0, otherwise SP
                    \[d\] = Rmode(m, M32_Undef);
                when '1110x'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    m = 14 - UInt(SYSm<0>);  // LR when SYSm<0> == 0, otherwise SP
                    \[d\] = Rmode(m, M32_Monitor);
                when '11110'  \[d\] = ELR_hyp;
                when '11111'  \[d\] = Rmode[13, M32_Hyp];
        end;
end;

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**MSR (Banked register)**

Move to Banked or Special register from general-purpose register moves the value of a general-purpose register to the Banked general-purpose register or Saved Program Status Registers (SPSRs) of the specified mode, or to ELR_hyp. MSR (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MSR (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MSR (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the Banked register transfer instructions*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```assembly
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>
```

**cond**

*MSR*{<c>}{<q>} <banked_reg>, <Rn>

n = UInt(Rn); write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE;
SYSm = M:M1;

### T1

```assembly
<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1 0 0</td>
</tr>
</tbody>
</table>
```

*MSR*{<c>}{<q>} <banked_reg>, <Rn>

n = UInt(Rn); write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
SYSm = M:M1;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<banked_reg>` Is the name of the banked register to be transferred to or from, encoded in “R:M:M1”:

---

**MSR (Banked register)**

Page 275
<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
<th>&lt;banked_reg&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>R8_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>R9_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0010</td>
<td>R10_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0011</td>
<td>R11_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0100</td>
<td>R12_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0101</td>
<td>SP_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0110</td>
<td>LR_usr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0111</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>R8_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1001</td>
<td>R9_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1010</td>
<td>R10_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1011</td>
<td>R11_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1100</td>
<td>R12_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>SP_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1110</td>
<td>LR_fiq</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1111</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0000</td>
<td>LR_irq</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0001</td>
<td>SP_irq</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0010</td>
<td>LR_svc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0011</td>
<td>SP_svc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0100</td>
<td>LR_abt</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0101</td>
<td>SP_abt</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0110</td>
<td>LR_und</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0111</td>
<td>SP_und</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10xx</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1100</td>
<td>LR_mon</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1101</td>
<td>SP_mon</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1110</td>
<td>ELR_hyp</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>SP_hyp</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0xxx</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10xx</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110x</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1110</td>
<td>SPSR_fiq</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1111</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0000</td>
<td>SPSR_irq</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0010</td>
<td>SPSR_svc</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0011</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0100</td>
<td>SPSR_abt</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0101</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0110</td>
<td>SPSR_und</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0111</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10xx</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1100</td>
<td>SPSR_mon</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1101</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1110</td>
<td>SPSR_hyp</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>UNPREDICTABLE</td>
</tr>
</tbody>
</table>

<Rn> Is the general-purpose source register, encoded in the "Rn" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
else
    mode = PSTATE.M;
    if write_spsr then
        SPSRAccessValid(SYSm, mode);  // Check for UNPREDICTABLE cases
        case SYSm of
            when '01110'  SPSR_fiq = ZeroExtend(R[n]);
            when '10000'  SPSR_irq = ZeroExtend(R[n]);
            when '10010'  SPSR_svc = ZeroExtend(R[n]);
            when '10100'  SPSR_abt = ZeroExtend(R[n]);
            when '10110'  SPSR_und = ZeroExtend(R[n]);
            when '11100'
                if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                SPSR_mon = R[n];
            when '11110'  SPSR_hyp = R[n];
        else
            BankedRegisterAccessValid(SYSm, mode);  // Check for UNPREDICTABLE cases
            case SYSm of
                when '00xxx'                       // Access the User mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    Rmode[m,M32_User] = R[n];
                when '01xxx'                       // Access the FIQ mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    Rmode[m,M32_FIQ] = R[n];
                when '1000x'                       // Access the IRQ mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_IRQ] = R[n];
                when '1001x'                       // Access the Supervisor mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Svc] = R[n];
                when '1010x'                       // Access the Abort mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Abort] = R[n];
                when '1011x'                       // Access the Undefined mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Undef] = R[n];
                when '1110x'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    Rmode[m,M32_Monitor] = R[n];
                when '11110'  // Access Monitor registers
                    ELR_hyp = R[n];
                when '11111'  // Access SP_hyp register
                    Rmode[13,M32_Hyp] = R[n];
        endcase
    endif
endif

CONSTRUCTED UNPREDICTABLE behavior

If PSTATE.EL == EL0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MSR (immediate)

Move immediate value to Special register moves selected bits of an immediate value to the corresponding bits in the APSR, CPSR, or SPSR <current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, the immediate form of MSR is normally only useful at the Application level for writing to APSR_nzcvq (CPSR_f).

If an MSR (immediate) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See Illegal changes to PSTATE.M.

An MSR (immediate) executed in User mode:
- Is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher,

An MSR (immediate) executed in System mode is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an MSR instruction. Arm deprecates using this to change its value.

A1

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>1111</td>
<td>0 0</td>
<td>1 1</td>
<td>0</td>
<td>R</td>
<td>1 0</td>
<td>mask</td>
<td>[(1)(1)(1)(1)]</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

cond

A1 (!(R == 0 && mask == 0000))

```
MSR{<c>}{<q>}, #<imm>
```

if mask == '0000' && R == '0' then SEE "Related encodings";
imm32 = A32ExpandImm(imm12); write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If mask == '0000' && R == '1', then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as NOP.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: Move Special Register and Hints (immediate).

**Assembler Symbols**

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<spec_reg> Is one of:
- APSR_<bits>,
- CPSR_<fields>,
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

- mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.
- mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.
- mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.
- mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of nzcvq, g, or nzcvqg. These map to the following CPSR_<fields> values:
• APSR\_nzcvq is the same as CPSR\_f (mask == '1000').
• APSR\_g is the same as CPSR\_s (mask == '0100').
• APSR\_nzcvqg is the same as CPSR\_fs (mask == '1100').

Arm recommends the APSR\_<bits>\> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see The Application Program Status Register, APSR.

<imm> is an immediate value. See Modified immediate constants in A32 instructions for the range of values.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
else if write_spsr then
    if PSTATE.M IN {M32\_User, M32\_System} then
        UNPREDICTABLE;
    else
        SPSRWriteByInstr(imm32, mask);
else
    // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
    CPSRWriteByInstr(imm32, mask);
```

**CONSTRAINED UNPREDICTABLE behavior**

If PSTATE.M IN {M32\_User, M32\_System} && write_spsr, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
MSR (register)

Move general-purpose register to Special register moves selected bits of a general-purpose register to the APSR, CPSR or SPSR `<current_mode>`.

Because of the Do-Not-Modify nature of its reserved bits, a read-modify-write sequence is normally required when the MSR instruction is being used at Application level and its destination is not APSR_nzcvq (CPSR_f).

If an MSR (register) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See Illegal changes to PSTATE.M.

An MSR (register) executed in User mode:

- Is UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher.

An MSR (register) executed in System mode is UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an MSR instruction. Arm deprecates using this to change its value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond != 1111 | 0 0 0 1 0 | R | 1 | 0 | mask |(1)(1)(1)(1)|(0)|0|0|0|0|0|0|0|0|0|0|0|0|0|Rn
```

### A1

```
MSR{<c>}{<q}> <spec_reg>, <Rn>
```

```
n = UInt(Rn); write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If mask == '0000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

### T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond
```

### T1

```
MSR{<c>}{<q}> <spec_reg>, <Rn>
```

```
n = UInt(Rn); write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

### CONSTRAINED UNPREDICTABLE behavior

If mask == '0000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.
<spec_reg> Is one of:
- APSR_<bits>.
- CPSR_<fields>.
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

c mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.

x mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.

s mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.

f mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of nzcvq, g, or nzcvqg. These map to the following CPSR_<fields> values:
- APSR_nzcvq is the same as CPSR_f (mask== '1000').
- APSR_g is the same as CPSR_s (mask == '0100').
- APSR_nzcvqg is the same as CPSR_fs (mask == '1100').

Arm recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *The Application Program Status Register, APSR*.

<Rn> Is the general-purpose source register, encoded in the “Rn” field.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User, M32_System} then
            UNPREDICTABLE;
        else
            SPSRWriteByInstr(R[n], mask);
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(R[n], mask);
```

CONstrained UNPREdictable behavior

If write_spsr & PSTATE.M IN {M32_User, M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
**MUL, MULS**

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Optionally, it can update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| != 1111 0 0 0 0 0 0 0 0 S       | Rd                               | (0) (0) (0) (0)                 | Rm                               | 1 0 0 1 Rn                      |
| cond                                          |                                 |                                 |                                 |                                 |

**Flag setting (S == 1)**

MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}

**Not flag setting (S == 0)**

MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S == '1');
\]

if \(d == 15 \text{ || } n == 15 \text{ || } m == 15\) then UNPREDICTABLE;

### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 0 1 0 0 0 0 1 1 0 1             | Rn                               | Rdm                             |

**T1**

MUL{<c>}{<q>} <Rdm>, <Rn>{, <Rdm>} // (Inside IT block)

MUL{<q>} <Rdm>, <Rn>{, <Rdm>} // (Outside IT block)

\[
d = \text{UInt}(Rdm); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rdm); \quad \text{setflags} = !\text{InITBlock}();
\]

### T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 1 1 1 0 1 1 0 0 0 0       | Rn                               | 1 1 1 1                        | Rd                               | 0 0 0 0 Rm                      |

**T2**

MUL{<c>}.W <Rd>, <Rn>{, <Rm>} // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}

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

if \(d == 15 \text{ || } n == 15 \text{ || } m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
<q> See Standard assembler syntax fields. </q>

<Rdm> Is the second general-purpose source register holding the multiplier and the destination register, encoded in the “Rdm” field. </Rdm>

<Rd> Is the general-purpose destination register, encoded in the “Rd” field. </Rd>

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field. </Rn>

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field. If omitted, <Rd> is used.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**MVN, MVNS (immediate)**

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register.
If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

cond
```

**MVN (S == 0)**

```
MVN{<c>}{<q>} <Rd>, #<const>
```

**MVNS (S == 1)**

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

```python
d = UInt(Rd); setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

**T1**

```
0 1 1 1 0 | i | 0 0 0 1 1 | S | 1 1 1 1 0 |
```

**MVN (S == 0)**

```
MVN{<c>}{<q>} <Rd>, #<const>
```

**MVNS (S == 1)**

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

```python
d = UInt(Rd); setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.

For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    result = NOT(imm32);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
        // PSTATE.V unchanged
```
MVN, MVNS (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register. If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
</tr>
<tr>
<td><strong>cond</strong></td>
</tr>
<tr>
<td>MVN{&lt;c}&gt;{&lt;q&gt;} &lt;Rd&gt;, &lt;Rm&gt;, RRX</td>
</tr>
</tbody>
</table>

MVN, shift or rotate by value (S == 0 & !(imm5 == 00000 & stype == 11))

MVN{<c}>{<q>} <Rd>, <Rm> {, <shift> #<amount>}

MVNS, rotate right with extend (S == 1 & imm5 == 00000 & stype == 11)

MVNS{<c}>{<q>} <Rd>, <Rm>, RRX

MVNS, shift or rotate by value (S == 1 & !(imm5 == 00000 & stype == 11))

MVNS{<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>}

d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 1 1 1 1</td>
</tr>
</tbody>
</table>

MVN{<c>|<q>} <Rd>, <Rm> // (Inside IT block)

MVNS{<q>} <Rd>, <Rm> // (Outside IT block)

d = UInt(Rd); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);

T2
### MVN, MVNS (register)

<table>
<thead>
<tr>
<th>-imm3</th>
<th>Rd</th>
<th>imm2</th>
<th>stype</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

MVN, rotate right with extend ($S == 0 \&\& \text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11$)

\[
\text{MVN}\{<c>\}\{<q>\} \quad \text{<Rd>}, \quad \text{<Rm>}, \quad \text{RRX}
\]

MVN, shift or rotate by value ($S == 0 \&\& !\text{(imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11))$

\[
\text{MVN}<c>.W \quad \text{<Rd>}, \quad \text{<Rm>} \quad // \quad \text{(Inside IT block, and <Rd>, <Rm> can be represented in T1)}
\]

\[
\text{MVN}\{<c>\}\{<q>\} \quad \text{<Rd>}, \quad \text{<Rm>} \quad \{, \quad \text{<shift>} \quad \#\text{<amount>}}
\]

MVNS, rotate right with extend ($S == 1 \&\& \text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11$)

\[
\text{MVNS}\{<c>\}\{<q>\} \quad \text{<Rd>}, \quad \text{<Rm>}, \quad \text{RRX}
\]

MVNS, shift or rotate by value ($S == 1 \&\& !\text{(imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11))$

\[
\text{MVNS}.W \quad \text{<Rd>}, \quad \text{<Rm>} \quad // \quad \text{(Outside IT block, and <Rd>, <Rm> can be represented in T1)}
\]

\[
\text{MVNS}\{<c>\}\{<q>\} \quad \text{<Rd>}, \quad \text{<Rm>} \quad \{, \quad \text{<shift>} \quad \#\text{<amount>}}
\]

\[
d = \text{UInt}(\text{Rd}); \quad m = \text{UInt}(\text{Rm}); \quad \text{setflags} = (S == '1');
\]

\[
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype, imm3:imm2});
\]

\[
\text{if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

#### Assembler Symbols

* `<c>` See Standard assembler syntax fields.

* `<q>` See Standard assembler syntax fields.

* `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:
  * For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  * For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

  For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field.

* `<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

  For encoding T1 and T2: is the general-purpose source register, encoded in the "Rm" field.

* `<shift>` Is the type of shift to be applied to the source register, encoded in "stype":

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

* `<amount>` For encoding A1: is the shift amount, in the range 1 to 31 (when `<shift> = LSL or ROR) or 1 to 32 (when `<shift> = LSR or ASR), encoded in the "imm5" field as `<amount>` modulo 32.

  For encoding T2: is the shift amount, in the range 1 to 31 (when `<shift> = LSL or ROR) or 1 to 32 (when `<shift> = LSR or ASR), encoded in the "imm3:imm2" field as `<amount>` modulo 32.
if `ConditionPassed()` then
    EncodingSpecificOperations();
    (shifted, carry) = `Shift_C(R[m], shift_t, shift_n, PSTATE.C)`;
    result = NOT(shifted);
    if `d == 15` then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        `R[d] = result`;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged

MVN, MVNS (register)
MVN, MVNS (register-shifted register)

Bitwise NOT (register-shifted register) writes the bitwise inverse of a register-shifted register value to the destination register. It can optionally update the condition flags based on the result.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rd</td>
<td>Rs</td>
<td>0</td>
<td>stype</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag setting (S == 1)

MVNS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

Not flag setting (S == 0)

MVN{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

\[\begin{align*}
\text{cond} & = \text{S == '1'}; \\
\text{setflags} & = \text{S == '1'}; \\
\text{shift_t} & = \text{DecodeRegShift(stype)};
\end{align*}\]

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the type of shift to be applied to the second source register, encoded in "stype":

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

- `<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```
NOP

No Operation does nothing. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

**A1**

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

cond
```

NOP{<c>}{<q>}

// No additional decoding required

**T1**

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
```

T1 NOP{<c>}{<q>}

// No additional decoding required

**T2**

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(1)(1)(1)(1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

T2 NOP{<c>}.W

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.

**Operation**

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to 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, ORNS (immediate)

Bitwise OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S</td>
<td>!=</td>
<td>1111</td>
<td>0</td>
<td>imm3</td>
<td>Rd</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm8</td>
</tr>
</tbody>
</table>

Flag setting (S == 1)

ORN{<c>}{<q>} {<Rd>,} <Rn>, #<const>

Not flag setting (S == 0)

ORN{<c>}{<q>} {<Rd>,} <Rn>, #<const>

if Rn == '1111' then SEE "MVN (immediate)"

\[
\begin{align*}
\text{d} &= \text{UInt}(\text{Rd}); \\
\text{n} &= \text{UInt}(\text{Rn}); \\
\text{setflags} &= (S == '1'); \\
(\text{imm32}, \text{carry}) &= \text{T32ExpandImm_C}(i:imm3:imm8, \text{PSTATE}.C); \\
\text{if d == 15 then UNPREDICTABLE;} // \text{Armv8-A removes UNPREDICTABLE for R13}
\end{align*}
\]

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

<const> An immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  result = R[n] OR NOT(imm32);
  R[d] = result;
  if setflags then
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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, ORNS (register)

Bitwise OR NOT (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. It can optionally update the condition flags based on the result.

| T1 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | S | != | 1111 | (0) | imm3 | Rd | imm2 | stype | Rm |

ORN, rotate right with extend (S == 0 & imm3 == 000 & imm2 == 00 & stype == 11)

ORN{<c>}{<q>}{<Rd>}, {<Rn>}, {<Rm>}, RRX

ORN, shift or rotate by value (S == 0 & !(imm3 == 000 & imm2 == 00 & stype == 11))

ORN{<c>}{<q>}{<Rd>}, {<Rn>}, {<Rm>}, {<shift> #<amount>}

ORN{<c>}{<q>}{<Rd>}, {<Rn>}, {<Rm>}, RRX

ORN{<c>}{<q>}{<Rd>}, {<Rn>}, {<Rm>}, {<shift> #<amount>}

if Rn == '1111' then SEE "MVN (register)";

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in "stype":

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

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.
Operation

if `ConditionPassed()` then
    EncodingSpecificOperations();
    (shifted, carry) = `Shift_C(R[m], shift_t, shift_n, PSTATE.C);`
    result = `R[n] OR NOT(shifted);`
    `R[d] = result;`
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = `IsZeroBit(result);`
        PSTATE.C = carry;
        // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

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

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------------|
| != 1111 0 0 1 1 1 0 0 S | Rx | Rd | imm12 |

cond

ORR (S == 0)

ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>

ORRS (S == 1)

ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

\[ d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1'); \]

\[ (\text{imm32}, \text{carry}) = \text{A32ExpandImm}_C(\text{imm12}, \text{PSTATE}.C); \]

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------------|
| 1 1 1 1 0 | i | 0 0 0 1 0 S | != 1111 0 | imm3 | Rd | imm8 |

ORR (S == 0)

ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>

ORRS (S == 1)

ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

if Rn == '1111' then SEE "MOV (immediate)";
\[ d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1'); \]

\[ (\text{imm32}, \text{carry}) = \text{T32ExpandImm}_C(i:mm3:mm8, \text{PSTATE}.C); \]

if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c>  See Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
   • For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
   • For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

if ConditionPassed() then
   EncodingSpecificOperations();
   result = R[n] OR imm32;
   if d == 15 then          // Can only occur for A32 encoding
      if setflags then
         ALUExceptionReturn(result);
      else
         ALUWritePC(result);
   else
      R[d] = result;
      if setflags then
         PSTATE.N = result<31>;
         PSTATE.Z = IsZeroBit(result);
         PSTATE.C = carry;
         // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
   • The execution time of this instruction is independent of:
      ◦ The values of the data supplied in any of its registers.
      ◦ The values of the NZCV flags.
   • The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ORR, ORRS (register)**

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

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| != 1111 | 0 0 0 1 1 0 0 | S | Rn | Rd | imm5 | stype | 0 | Rm |

cond

**ORR, rotate right with extend (S == 0 & amp; amp; imm5 == 00000 & amp; & stype == 11)**

ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**ORR, shift or rotate by value (S == 0 & & !(imm5 == 00000 & & stype == 11))**

ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

**ORRS, rotate right with extend (S == 1 & & imm5 == 00000 & & stype == 11)**

ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**ORRS, shift or rotate by value (S == 1 & & !(imm5 == 00000 & & stype == 11))**

ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S == '1');
\]

\[
\text{(shift}_t, \text{shift}_n) = \text{DecodeImmShift(stype, imm5)};
\]

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------------------------|-----------|-----------|
| 0 1 0 0 0 0 0 1 1 0 0 | Rm | Rdn |

**T1**

ORR{<c>}{<q>} {<Rdn>,} <Rdn>, <Rm> // (Inside IT block)

ORRS{<c>} {<Rdn>,} <Rdn>, <Rm> // (Outside IT block)

\[
d = \text{UInt}(Rdn); \quad n = \text{UInt}(Rdn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = !\text{InITBlock}();
\]

\[
\text{(shift}_t, \text{shift}_n) = (\text{SRTypELSL}, 0);
\]

**T2**
ORR, rotate right with extend (S == 0 & imm3 == 000 & imm2 == 00 & stype == 11)

ORR{<c>{<q>}{<Rd>,}{<Rn>,}{<Rm>}, RRX}

ORR, shift or rotate by value (S == 0 & !(imm3 == 000 & imm2 == 00 & stype == 11))

ORR<c>.W (<Rd>,) <Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
ORR{<c>{<q>}{<Rd>,}{<Rn>,}{<Rm>}, , <shift> #{amount>}

ORRS, rotate right with extend (S == 1 & imm3 == 000 & imm2 == 00 & stype == 11)

ORRS{<c>{<q>}{<Rd>,}{<Rn>,}{<Rm>}, RRX}

ORRS, shift or rotate by value (S == 1 & !(imm3 == 000 & imm2 == 00 & stype == 11))

ORRS.W (<Rd>,) <Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
ORRS{<c>{<q>}{<Rd>,}{<Rn>,}{<Rm>}, , <shift> #{amount>}

if Rn == '1111' then SEE "Related encodings";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Related encodings: Data-processing (shifted register)

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  - For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in "stype":

ORR, ORRS (register)
### stype | <shift>
---|---
00 | LSL
01 | LSR
10 | ASR
11 | ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

In T32 assembly:
- Outside an IT block, if ORRS <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORRS <Rd>, <Rn> had been written.
- Inside an IT block, if ORR<c> <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORR<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

#### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR shifted;
    if d == 15 then  // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

#### Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

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

Bitwise OR (register-shifted register) performs a bitwise (inclusive) OR of a register value and a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

<table>
<thead>
<tr>
<th>!= 1111</th>
<th>0 0 0 1 1 0 0 S</th>
<th>Rn</th>
<th>Rd</th>
<th>Rs</th>
<th>0</th>
<th>stype</th>
<th>1</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag setting (S == 1)

ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

Not flag setting (S == 0)

ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in "stype":

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

<Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  shift_n = UInt(R[s]<7:0>);
  (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
  result = R[n] OR shifted;
  R[d] = result;
  if setflags then
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccccccccc}
\hline
!= 1111 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & Rn & Rd & \text{imm5} & \text{tb} & 0 & 1 & \text{Rm}
\end{array}
\]

PKHBT (tb == 0)

PKHBT\{<c>\}{<q>\} \{<Rd>, \} <Rn>, <Rm> \{, LSL \#<imm>\}

PKHBT (tb == 1)

PKHBT\{<c>\}{<q>\} \{<Rd>, \} <Rn>, <Rm> \{, ASR \#<imm>\}

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ tbform = (tb == '1');
\]

\[
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(tb:'0', \text{imm5});
\]

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

\[
\begin{array}{cccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & \text{Rn} \{0\} & \text{imm3} & \text{Rd} & \text{imm2} & \text{tb} & 0 & \text{Rm}
\end{array}
\]

PKHBT (tb == 0)

PKHBT\{<c>\}{<q>\} \{<Rd>, \} <Rn>, <Rm> \{, LSL \#<imm>\} // (tbform == FALSE)

PKHBT (tb == 1)

PKHBT\{<c>\}{<q>\} \{<Rd>, \} <Rn>, <Rm> \{, ASR \#<imm>\} // (tbform == TRUE)

if S == '1' || T == '1' then UNDEFINED;

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ tbform = (tb == '1');
\]

\[
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(tb:'0', \text{imm3:imm2});
\]

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

\(<\text{imm}>\) For encoding A1: the shift to apply to the value read from \(<Rm>\), encoded in the "imm5" field.

For PKHBT, it is one of:

\text{omitted}

No shift, encoded as 0b00000.
1-31
Left shift by specified number of bits, encoded as a binary number.

For PKHTB, it is one of:

**omitted**
Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>{<q}> <Rd>, <Rm>, <Rn> had been written.

1-32
Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.

An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

For encoding T1: the shift to apply to the value read from <Rm>, encoded in the “imm3:imm2” field.
For PKHBT, it is one of:

**omitted**
No shift, encoded as 0b00000.

1-31
Left shift by specified number of bits, encoded as a binary number.

For PKHTB, it is one of:

**omitted**
Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>{<q}> <Rd>, <Rm>, <Rn> had been written.

1-32
Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.

An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

**Operation**

if **ConditionPassed**() then

```
EncodingSpecificOperations();
operand2 = Shift(R[m], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
R[d]<15:0> = if tbform then operand2<15:0> else R[n]<15:0>;
R[d]<31:16> = if tbform then R[n]<31:16> else operand2<31:16>;
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Preload Data (immediate) signals the memory system that data memory accesses from a specified address are likely 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 the data cache. The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write. The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

**A1**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | U  | R  | 0  | 1  | ! = 1111 | (1)(1)(1)(1) | imm12 |
| Rn |

**Preload read (R == 1)**

PLD{<c>}{<q>} [{<Rn}> {, #{+/-}<imm>}]}

**Preload write (R == 0)**

PLDW{<c>}{<q>} [{<Rn}> {, #{+/-}<imm>}]}

if Rn == '1111' then SEE "PLD (literal)";

n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); add = (U == '1'); is_pldw = (R == '0');

**T1**

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>W</td>
<td>1</td>
<td>! = 1111</td>
<td>1 1 1 1</td>
<td>imm12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Preload read (W == 0)**

PLD{<c>}{<q>} [{<Rn}> {, #<+><imm>}]}

**Preload write (W == 1)**

PLDW{<c>}{<q>} [{<Rn}> {, #<+><imm>}]}

if Rn == '1111' then SEE "PLD (literal)";

n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); add = TRUE; is_pldw = (W == '1');

**T2**

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>W</td>
<td>1</td>
<td>! = 1111</td>
<td>1 1 1 1</td>
<td>1 1 0 0</td>
<td>imm8</td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Preload read \((W == 0)\)

\[
\text{PLD\{<c>\}{<q>}} \ [\text{<Rn> \{, \#-<imm>\}}]
\]

Preload write \((W == 1)\)

\[
\text{PLDW\{<c>\}{<q>}} \ [\text{<Rn> \{, \#-<imm>\}}]
\]

if Rn == '1111' then SEE "PLD (literal)";

\[n = \text{UInt}(\text{Rn}); \text{ imm32} = \text{ZeroExtend}(\text{imm8, 32}); \text{ add} = \text{FALSE}; \text{ is_pldw} = (W == '1');\]

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\)

For encoding A1: see Standard assembler syntax fields. Must be AL or omitted.

For encoding T1 and T2: see Standard assembler syntax fields.

\(<q>\)

See Standard assembler syntax fields.

\(<\text{Rn}>\)

Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see PLD (literal).

\(+/-\)

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th></th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

+  

Specifies the offset is added to the base register.

\(<\text{imm}>\)

For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation

if ConditionPassed() then

    EncodingSpecificOperations();
    address = if add then \((R[n] + \text{imm32})\) else \((R[n] - \text{imm32})\);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);


Preload Data (literal) signals the memory system that data memory accesses from a specified address are likely 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 the data cache. The effect of a PLD instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**Assembler Symbols**

<

For encoding A1: see Standard assembler syntax fields. Must be AL or omitted.

For encoding T1: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<label>

The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range –4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

+/-(U)

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

<imm>

For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.
The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Preload Data (register) signals the memory system that data memory accesses from a specified address are likely 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 the data cache. The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 1  | 1  | U | R | 0 | 1 | Rn | (1)(1)(1)(1) | imm5 | stype | 0 | Rm |

Preload read, optional shift or rotate (R == 1 & (imm5 == 00000 & stype == 11))

PLD{<c>}{<q>} [ <Rn>, {+/-}<Rm> {, <shift> #<amount>}]

Preload read, rotate right with extend (R == 1 & imm5 == 00000 & stype == 11)

PLD{<c>}{<q>} [ <Rn>, {+/-}<Rm> , RRX]

Preload write, optional shift or rotate (R == 0 & (imm5 == 00000 & stype == 11))

PLDW{<c>}{<q>} [ <Rn>, {+/-}<Rm> {, <shift> #<amount>}]

Preload write, rotate right with extend (R == 0 & imm5 == 00000 & stype == 11)

PLDW{<c>}{<q>} [ <Rn>, {+/-}<Rm> , RRX]

n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');  is_pldw = (R == '0');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 || (n == 15 & is_pldw) then UNPREDICTABLE;

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>W</td>
<td>1</td>
<td>!=</td>
<td>1111</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Preload read (W == 0)

PLD{<c>}{<q>} [ <Rn>, {+}<Rm> {, LSL #<amount>}]  

Preload write (W == 1)

PLDW{<c>}{<q>} [ <Rn>, {+}<Rm> {, LSL #<amount>}]  

if Rn == '1111' then SEE "PLD (literal)";

n = UInt(Rn);  m = UInt(Rm);  add = TRUE;  is_pldw = (W == '1');
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. <c> must be AL or omitted.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used.
For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

<+/-> Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted
and encoded in “U”:

<table>
<thead>
<tr>
<th></th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the index register, encoded in “stype”:

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

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32
(when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.
For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the “imm2”
field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
PLI (immediate, literal)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely 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 pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

A1

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

A1

PLI{<c>}{<q>} {<Rn> {, #<+/-><imm>}}

PLI{<c>}{<q>} <label> // (Normal form)

PLI{<c>}{<q>} {PC, #<+/-><imm>} // (Alternative form)

n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); add = (U == '1');

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>

if Rn == '1111' then SEE "encoding T3";

n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); add = TRUE;

T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 0 1 0 0 0 1</td>
</tr>
</tbody>
</table>

if Rn == '1111' then SEE "encoding T3";

n = UInt(Rn); imm32 = ZeroExtend(imm8, 32); add = FALSE;

T3

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 0 1</td>
</tr>
</tbody>
</table>
PLI\{<c}\}{<q> \label> // (Preferred syntax)

PLI\{<c}\}{<q> [PC, #\{+/-\}imm] // (Alternative syntax)

n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields. Must be AL or omitted.

For encoding T1, T2 and T3: see Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<\text{label}>\) The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range –4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

\(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

\[
\begin{array}{c|c}
  U & +/- \\
  \hline
  0 & - \\
  1 & +
\end{array}
\]

\(+\) Specifies the offset is added to the base register.

\(<\text{imm}>\) For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

For encoding T3: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For the literal forms of the instruction, encoding T3 is used, or Rn is encoded as 0b1111 in encoding A1, to indicate that the PC is the base register.

The alternative literal syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax.

Operation

if \text{ConditionPassed}() then
   EncodingSpecificOperations();
   base = if n == 15 then Align(PC,4) else R[n];
   address = if add then (base + imm32) else (base - imm32);
   Hint_PreloadInstr(address);
Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely 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 pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 1 1 | U | 1 0 | 1 | Rn | (1)(1)(1)(1) | imm5 | stype | 0 | Rm
```

#### Rotate right with extend (imm5 == 00000 && stype == 11)

```
PLI{<c>}{<q>} [ <Rn>, {+/-}<Rm> , RRX]
```

### T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 0 0 1 0 0 1 1 != 1111 | 1 1 1 1 0 0 0 0 0 0 | imm2 | Rm
```

#### Shift or rotate by value (!!(imm5 == 00000 && stype == 11))

```
PLI{<c>}{<q>} [ <Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

\[ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ \text{add} = (U == '1'); \]
\[ (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5}); \]
\[ \text{if } m == 15 \text{ then UNPREDICTABLE;} \]

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler Symbols

- `<c>` For encoding A1: see Standard assembler syntax fields. `<c>` must be AL or omitted.
- `<q>` See Standard assembler syntax fields.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
- `+/` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

+ Specifies the index register is added to the base register.
<Rm> Is the general-purpose index register; encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the index register, encoded in "stype":

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

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
```
**POP**

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td></td>
<td>register_list</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T1**

```
POP{<c>}{<q>} <registers> // (Preferred syntax)
LDM{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

```c
registers = P:'0000000':register_list; UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the **CONSTRAINED UNPREDICTABLE behavior** of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<registers>` Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R7, encoded in the "register list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0.

If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.
if `ConditionPassed()` then
  EncodingSpecificOperations();
  address = `SP`;
  for i = 0 to 14
    if registers<i> == '1' then
      `R[i]` = if UnalignedAllowed then `MemU[address,4]` else `MemA[address,4]`;
      address = address + 4;
  if registers<15> == '1' then
    if UnalignedAllowed then
      if address<1:0> == '00' then
        `LoadWritePC(MemU[address,4]);`
      else
        UNPREDICTABLE;
    else
      `LoadWritePC(MemA[address,4]);`
  if registers<13> == '0' then `SP` = `SP` + 4*`BitCount`(registers);
  if registers<13> == '1' then `SP` = bits(32) UNKNOWN;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
POP (multiple registers)

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

This is an alias of LDM, LDMIA, LDMFD. This means:

- The encodings in this description are named to match the encodings of LDM, LDMIA, LDMFD.
- The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 1111 1 0 0 0 1 0 1 1 1 1 0 1</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

A1

POP{<c>}{<q>} <registers>

is equivalent to

LDM{<c>}{<q>} SP!, <registers>

and is the preferred disassembly when BitCount(register_list) > 1.

T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 0 0 1 0 1 1 1 1 0 1</td>
</tr>
<tr>
<td>W</td>
</tr>
</tbody>
</table>

T2

POP{<c>}.W <registers> // (All registers in R0-R7, PC)

POP{<c>}{<q>} <registers>

is equivalent to

LDM{<c>}{<q>} SP!, <registers>

and is the preferred disassembly when BitCount(P:M:register_list) > 1.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<registers>` For encoding A1: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.
  If the SP is in the list, the value of the SP after such an instruction is UNKNOWN.
  The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  Arm deprecates the use of this instruction with both the LR and the PC in the list.
  For encoding T2: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the
highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

**Operation**

The description of **LDM, LDMIA, LDMFD** gives the operational pseudocode for this instruction.
POP (single register)

Pop Single Register from Stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point just above the loaded data.

This is an alias of LDR (immediate). This means:

- The encodings in this description are named to match the encodings of LDR (immediate).
- The description of LDR (immediate) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T4).

A1

\[
\begin{array}{cccccccccccccccccc}
\hline
=! & 1111 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

Post-indexed

POP\{<c>\}{<q>} <single_register_list>

is equivalent to

LDR\{<c>\}{<q>} <Rt>, [SP], #4

and is always the preferred disassembly.

T4

\[
\begin{array}{cccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

Post-indexed

POP\{<c>\}{<q>} <single_register_list>

is equivalent to

LDR\{<c>\}{<q>} <Rt>, [SP], #4

and is always the preferred disassembly.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<\text{single_register_list}>\) Is the general-purpose register \(<\text{Rt}>\) to be loaded surrounded by \{ and \}.

\(<\text{Rt}>\) For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
**Operation**

The description of **LDR (immediate)** gives the operational pseudocode for this instruction.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.

- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & (1) & (1) & (1) & (1) & (1) & (1) & (0) & (0) & (0) & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
\]

T1

\[
\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & (1) & (1) & (1) & 1 & 0 & (0) & 0 & (1) & (1) & (1) & (1) & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
\]

Assembler Symbols

\(<q>\)  See Standard assembler syntax fields.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  SpeculativeStoreBypassBarrierToPA();
PUSH

Push Multiple Registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC.*

T1

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 & M & \text{register_list}
\end{array}
\]

T1

\[\text{PUSH}\{<c>\}{<q>} \text{ <registers>} // (Preferred syntax)\]
\[\text{STMDB}\{<c>\}{<q>} \text{ SP!, <registers>} // (Alternate syntax)\]

\[
\text{registers} = '0':M:'000000':\text{register_list}; \text{ UnalignedAllowed} = \text{FALSE};
\]
\[\text{if BitCount}(\text{registers}) < 1 \text{ then UNPREDICTABLE};\]

**CONSTRAINED UNPREDICTABLE behavior**

If BitCount(\text{registers}) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors.*

**Assembler Symbols**

\(<c>\) See *Standard assembler syntax fields.*

\(<q>\) See *Standard assembler syntax fields.*

\(<\text{registers}>\) Is a list of one or more registers to be stored, separated by commas and surrounded by \{ and \}. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0.
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == 13 && i != LowestSetBit(registers) then  // Only possible for encoding A1
                MemA[address,4] = bits(32) UNKNOWN;
            else
                if UnalignedAllowed then
                    MemU[address,4] = R[i];
                else
                    MemA[address,4] = R[i];
                address = address + 4;
        if registers<15> == '1' then  // Only possible for encoding A1 or A2
            if UnalignedAllowed then
                MemU[address,4] = PCStoreValue();
            else
                MemA[address,4] = PCStoreValue();
        SP = SP - 4*BitCount(registers);
**PUSH (multiple registers)**

Push multiple registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

This is an alias of STMDB, STMFD. This means:

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

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 1 0 0 1 0 0 1 0 1 1 0 1</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>
```

**A1**

```
PUSH{<c>{<q>} <registers>}
```

is equivalent to

```
STMDB{<c>{<q}>} SP!, <registers>
```

and is the preferred disassembly when BitCount(register_list) > 1.

**T1**

```
<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 0 1 0 1 0 1 0 1 1 0 1</td>
</tr>
<tr>
<td>W</td>
</tr>
</tbody>
</table>
```

**T1**

```
PUSH{<c>}.W <registers> // (All registers in R0-R7, LR)
PUSH{<c>{<q>} <registers>}
```

is equivalent to

```
STMDB{<c>{<q}>} SP!, <registers>
```

and is the preferred disassembly when BitCount(M:register_list) > 1.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<registers>` For encoding A1: is a list of two or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.
  
  The SP and PC can be in the list. However:
  - Arm deprecates the use of instructions that include the PC in the list.
  - If the SP is in the list, and it is not the lowest-numbered register in the list, the instruction stores an UNKNOWN value for the SP.

  For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the
highest-numbered register to the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

**Operation**

The description of [STMDB, STMFD](#) gives the operational pseudocode for this instruction.

---

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PUSH (single register)

Push Single Register to Stack stores a single general-purpose register to the stack, storing to the 32-bit word below the address in SP, and updates SP to point to the start of the stored data.

This is an alias of STR (immediate). This means:

- The encodings in this description are named to match the encodings of STR (immediate).
- The description of STR (immediate) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T4).

**A1**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|-------------------|---------------------|---------------------|-------------------|
| != 1111                                       | 0 1 0 1 0 1 1 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 |
| cond                                         | P U W              | Rn imm12            |
```

**Pre-indexed**

PUSH{<c>}{<q>} <single_register_list>

is equivalent to

STR{<c>}{<q>} <Rt>, [SP, #-4]!

and is always the preferred disassembly.

**T4**

```
| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|-------------------|---------------------|---------------------|-------------------|
| 1 1 1 1 1 0 0 0 0 1 0 0 1 1 0 1 1 0 1 0 0 0 0 0 0 1 0 0 |
| Rn                                         | P U W imm8        |
```

**Pre-indexed**

PUSH{<c>}{<q>} <single_register_list> // (Standard syntax)

is equivalent to

STR{<c>}{<q>} <Rt>, [SP, #-4]!

and is always the preferred disassembly.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<single_register_list>` Is the general-purpose register <Rt> to be stored surrounded by { and }.
- `<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
  For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

**Operation**

The description of STR (immediate) gives the operational pseudocode for this instruction.
**QADD**

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range \(-2^{31}\) to \((2^{31} - 1)\), and writes the result to the destination register. If saturation occurs, it sets \(\text{PSTATE.Q}\) to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

\[
\begin{array}{cccccccccccccccccccccccccccccc}
\hline
! = 1111 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & Rn & Rd & (0) & (0) & (0) & (0) & 0 & 1 & 0 & 1 & Rm \\
\end{array}
\]

\[
\text{cond}
\]

\[
\text{QADD}\{<c>\}{<q>} \{<Rd>,}\ <Rm>, <Rn>
\]

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

if \(d == 15 || n == 15 || m == 15\) then UNPREDICTABLE;

**T1**

\[
\begin{array}{cccccccccccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & Rn & 1 & 1 & 1 & Rd & 1 & 0 & 0 & 0 & Rm \\
\end{array}
\]

\[
\text{T1}
\]

\[
\text{QADD}\{<c>\}{<q>} \{<Rd>,\} <Rm>, <Rn>
\]

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

if \(d == 15 || n == 15 || m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- **<c>** See Standard assembler syntax fields.
- **<q>** See Standard assembler syntax fields.
- **<Rd>** Is the general-purpose destination register, encoded in the "Rd" field.
- **<Rm>** Is the first general-purpose source register, encoded in the "Rm" field.
- **<Rn>** Is the second general-purpose source register, encoded in the "Rn" field.

**Operation**

if \(\text{ConditionPassed()}\) then

\[
\text{EncodingSpecificOperations();}
\]

\[
(R[d], \text{sat}) = \text{SignedSatQ}(\text{SInt}(R[m])) + \text{SInt}(R[n]), 32);
\]

if sat then

\[
\text{PSTATE.Q} = '1';
\]

Internal version only: isa v01_19, pseudocode v2020-09 xml, sve v2020-09 rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
!= 1111 | 0 1 1 0 0 1 0 | Rn | Rd |(1)(1)(1)(1)| 0 | 0 | 1 | Rm
cond
```

A1

```plaintext
QADD16{{<c>}{<q}>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

**T1**

```plaintext
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 0 1 0 1 0 0 1 | Rn | 1 1 1 1 | Rd | 0 | 0 | 0 | 1 | Rm
```

T1

```plaintext
QADD16{{<c>}{<q}>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0> = SignedSat(sum1, 16);
    R[d]<31:16> = SignedSat(sum2, 16);
```
**QADD8**

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>cond</th>
<th>Rn</th>
<th>Rd</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 1 1 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A1**

**QADD8**{<c>}{<q>} {<Rd>,} {<Rn>,} {<Rm>}

```plaintext
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rn</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T1**

**QADD8**{<c>}{<q>} {<Rd>,} {<Rn>,} {<Rm>}

```plaintext
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the constrained UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

### Assembler Symbols

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0> = SignedSat(sum1, 8);
    R[d]<15:8> = SignedSat(sum2, 8);
    R[d]<23:16> = SignedSat(sum3, 8);
    R[d]<31:24> = SignedSat(sum4, 8);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range \(-2^{15} \leq x \leq 2^{15} - 1\), and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd | (1)(1)(1)(1) | 0 | 0 | 1 | 1 | Rm |

cond

A1

QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[
\text{if } d == 15 \lor n == 15 \lor m == 15 \text{ then UNPREDICTABLE;}
\]

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 0 | 0 | 1 | Rm |

T1

QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[
\text{if } d == 15 \lor n == 15 \lor m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then

EncodingSpecificOperations();

\[
\begin{align*}
\text{diff} & = \text{SInt}(R[n]<15:0>) - \text{SInt}(R[m]<31:16>); \\
\text{sum} & = \text{SInt}(R[n]<31:16>) + \text{SInt}(R[m]<15:0>); \\
R[d]<15:0> & = \text{SignedSat}(\text{diff}, 16); \\
R[d]<31:16> & = \text{SignedSat}(\text{sum}, 16);
\end{align*}
\]
QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range \(-2^{31} \leq x \leq 2^{31} - 1\). If saturation occurs in either operation, it sets \textit{PSTATE}.Q to 1.

It has encodings from the following instruction sets: A32 (\textit{A1}) and T32 (\textit{T1}).

**A1**

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

condition

\[
\begin{array}{cccccccccccccccccc}
! = 1111 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & Rn & Rd & (0)(0)(0)(0) & 0 & 1 & 0 & 1 & Rm
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
d = \text{UInt}(Rd); 
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
n = \text{UInt}(Rn); 
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
m = \text{UInt}(Rm); 
\end{array}
\]

if \(d == 15 \mid | n == 15 \mid | m == 15\) then UNPREDICTABLE;

**T1**

\[
\begin{array}{cccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & Rn & 1 & 1 & 1 & Rd & 1 & 0 & 0 & 1 & Rm
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
d = \text{UInt}(Rd); 
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
n = \text{UInt}(Rn); 
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
m = \text{UInt}(Rm); 
\end{array}
\]

if \(d == 15 \mid | n == 15 \mid | m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the \textit{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see \textit{Architectural Constraints on UNPREDICTABLE behaviors}.

**Assembler Symbols**

- \textit{<c>} See \textit{Standard assembler syntax fields}.
- \textit{<q>} See \textit{Standard assembler syntax fields}.
- \textit{<Rd>} Is the general-purpose destination register, encoded in the "Rd" field.
- \textit{<Rm>} Is the first general-purpose source register, encoded in the "Rm" field.
- \textit{<Rn>} Is the second general-purpose source register, encoded in the "Rn" field.

**Operation**

if \textit{ConditionPassed()} then

\[
\begin{array}{cccccccccccccccccc}
\text{EncodingSpecificOperations}(); 
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
(\text{doubled}, \text{sat1}) = \text{SignedSatQ}(2 \ast \text{SInt}(R[n]), 32); 
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
(\text{R}[d], \text{sat2}) = \text{SignedSatQ}(\text{SInt}(R[m]) + \text{SInt}(\text{doubled}), 32); 
\end{array}
\]

if \text{sat1} || \text{sat2} then

\[
\text{PSTATE}.Q = '1';
\]
QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range \(-2^{31} <= x <= 2^{31} - 1\). If saturation occurs in either operation, it sets `PSTATE.Q` to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| != 1111 | 0 0 0 1 0 1 1 0 | Rn | Rd | (0)|(0)|(0)|(0) | 0 1 0 1 | Rm |
```

cond

```c
QDSUB{<c>}{<q>}{<Rd>,} {<Rm>,} <Rn>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|----------------------------------------|
| 1 1 1 1 1 0 1 0 1 0 0 | Rn | 1 1 1 1 | Rd | 1 0 1 1 | Rm |
```

```c
QDSUB{<c>}{<q>}{<Rd>,} {<Rm>,} <Rn>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the constrained UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the first general-purpose source register, encoded in the "Rm" field.
- `<Rn>` Is the second general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    (R[d], sat2) = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range \(-2^{15} \leq x \leq 2^{15} - 1\), and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

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

if \(d == 15 \lor n == 15 \lor m == 15\) then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 0 1 1 1 0</td>
</tr>
</tbody>
</table>

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

if \(d == 15 \lor n == 15 \lor m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then

EncodingSpecificOperations();

\[
\begin{align*}
\text{sum} &= \text{SInt}(R[n]<15:0>) + \text{SInt}(R[m]<31:16>); \\
\text{diff} &= \text{SInt}(R[n]<31:16>) - \text{SInt}(R[m]<15:0>); \\
R[d]<15:0> &= \text{SignedSat}(\text{sum}, 16); \\
R[d]<31:16> &= \text{SignedSat}(\text{diff}, 16);
\end{align*}
\]

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**QSUB**

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range \(-2^{31} \leq x \leq 2^{31} - 1\), and writes the result to the destination register. If saturation occurs, it sets `PSTATE.Q` to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Rn | Rd | (0)|(0)|(0)|(0)| 0 | 1 | 0 | 1 | Rm |
```

```
cond
```

QSUB{<c>}{<q>} {<Rd>}, {<Rm>, <Rn>}

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

if \(d = 15 \text{ || } n = 15 \text{ || } m = 15\) then UNPREDICTABLE;

**T1**

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | 1  | 1  | 1 | 1  | 1  | 0  | 0 | 0 | Rn | 1 | 1 | 1 | 1 | Rd | 1 | 0 | 1 | 0 | Rm |
```

```
```

QSUB{<c>}{<q>} {<Rd>}, {<Rm>, <Rn>}

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

if \(d = 15 \text{ || } n = 15 \text{ || } m = 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the first general-purpose source register, encoded in the "Rm" field.
- `<Rn>` Is the second general-purpose source register, encoded in the "Rn" field.

**Operation**

if ConditionPassed() then

```
EncodingSpecificOperations();
(R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
if sat then
    PSTATE.Q = '1';
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range \(-2^{15} \leq x \leq 2^{15} - 1\), and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

cond

```plaintext
A1

QSUB16{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 0 1 1 0 1</td>
</tr>
</tbody>
</table>

```
T1

QSUB16{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

### Operation

if `ConditionPassed()` then

```plaintext
EncodingSpecificOperations();

diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
R[d]<15:0> = SignedSat(diff1, 16);
R[d]<31:16> = SignedSat(diff2, 16);
```
QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
1 1 1 1 1 1 1 1 | Rn | Rd |(1)| (1)| (1)| (1) | 1 1 1 1 1 | Rm
cond
```

A1

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

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

if \( d == 15 \) \(||\) \( n == 15 \) \(||\) \( m == 15 \) then UNPREDICTABLE;

T1

```
1 1 1 1 1 0 1 1 0 1 0 1 1 0 0 | Rn | 1 1 1 1 | Rd | 0 0 0 1 | Rm
```

T1

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

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

if \( d == 15 \) \(||\) \( n == 15 \) \(||\) \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the** constrained** UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

\(<c>\) See *Standard assembler syntax fields*.

\(<q>\) See *Standard assembler syntax fields*.

\(<\text{Rd}>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<\text{Rn}>\) Is the first general-purpose source register, encoded in the "Rn" field.

\(<\text{Rm}>\) Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

if \( \text{ConditionPassed}() \) then

\[
\text{EncodingSpecificOperations}();
\]

\[
diff1 = \text{SInt}(R[n]<7:0>) - \text{SInt}(R[m]<7:0>);
\]

\[
diff2 = \text{SInt}(R[n]<15:8>) - \text{SInt}(R[m]<15:8>);
\]

\[
diff3 = \text{SInt}(R[n]<23:16>) - \text{SInt}(R[m]<23:16>);
\]

\[
diff4 = \text{SInt}(R[n]<31:24>) - \text{SInt}(R[m]<31:24>);
\]

\[
R[d]<7:0> = \text{SignedSat}(\text{diff1}, 8);
\]

\[
R[d]<15:8> = \text{SignedSat}(\text{diff2}, 8);
\]

\[
R[d]<23:16> = \text{SignedSat}(\text{diff3}, 8);
\]

\[
R[d]<31:24> = \text{SignedSat}(\text{diff4}, 8);
\]
RBIT

Reverse Bits reverses the bit order in a 32-bit register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| != 1111 | 0 1 1 0 1 1 1 (1)(1)(1)| Rd | (1)(1)(1)(1) 0 0 1 1 | Rm |

cond

A1

RBIT{<c>}{<q>} <Rd>, <Rm>

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm);
\]
\[
\text{if } d == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 1 1 1 1 0 1 0 1 0 0 1 | Rn | 1 1 1 1 | Rd | 1 0 1 0 | Rm |

T1

RBIT{<c>}{<q>} <Rd>, <Rm>

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad n = \text{UInt}(Rn);
\]
\[
\text{if } m != n || d == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

CONSTRAINED UNPREDICTABLE behavior

If \( m \neq n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: \( m = \text{UInt}(Rn) \).
- The instruction executes with the additional decode: \( m = \text{UInt}(Rm) \).
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field.
For encoding T1: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    for i = 0 to 31
        result<31-i> = R[m]<i>;
    R[d] = result;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Byte-Reverse Word reverses the byte order in a 32-bit register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
! = 1111 | 0 1 1 0 1 0 1 1 | (1)(1)(1)(1) Rd | (1)(1)(1)(1) | 0 0 1 1 | Rm
cond
```

A1

```
REV{<c>}{<q>} <Rd>, <Rm>
```

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
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 | Rm | Rd
```

T1

```
REV{<c>}{<q>} <Rd>, <Rm>
```

```
d = UInt(Rd);  m = UInt(Rm);
```

T2

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

```
1 1 1 1 0 1 0 1 0 1 | Rn | 1 1 1 1 | Rd | 1 0 0 0 | Rm
```

T2

```
REV{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)
REV{<c>}{<q>} <Rd>, <Rm>
```

```
d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRANGED UNPREDICTABLE behavior

If m != n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: m = UInt(Rn);
- The instruction executes with the additional decode: m = UInt(Rm);
- The value in the destination register is UNKNOWN.

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

```
<c> See Standard assembler syntax fields.
```
See *Standard assembler syntax fields*.

**<Rd>**
Is the general-purpose destination register, encoded in the "Rd" field.

**<Rm>**
For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.
For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

### Operation

```python
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

### Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**REV16**

Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | (1)(1)(1)(1) | Rd | (1)(1)(1)(1) | 1 | 0 | 1 | 1 | Rm |
```

cond

### A1

```
REV16{<c>{cq}} <Rd>, <Rm>
```

d = UInt(Rd); m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;

### T1

```
| 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 | 1 | Rm | Rd |
```

### T1

```
REV16{<c>{cq}} <Rd>, <Rm>
```

d = UInt(Rd); m = UInt(Rm);

### T2

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | Rn | 1 | 1 | 1 | 1 | Rd | 1 | 0 | 0 | 1 | Rm |
```

### T2

```
REV16{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)
```

```
REV16{<c>}{cq} <Rd>, <Rm>
```

d = UInt(Rd); m = UInt(Rm); n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

#### CONSTRAINED UNPREDICTABLE behavior

If m != n, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes with the additional decode: m = UInt(Rn).
- The instruction executes with the additional decode: m = UInt(Rm).
- The value in the destination register is **UNKNOWN**.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

### Assembler Symbols

```
<c> See Standard assembler syntax fields.
```
<q> See Standard assembler syntax fields.
</q>

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

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

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign-extends the result to 32 bits.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

**A1**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>!= 1111</th>
<th>0 1 1 0 1</th>
<th>1 1 1</th>
<th>(1)(1)(1)(1)</th>
<th>Rd</th>
<th>(1)(1)(1)(1)(1)</th>
<th>1 0 1 1</th>
<th>Rm</th>
</tr>
</thead>
</table>
cond

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>1 0 1 1 1</th>
<th>0 1 1</th>
<th>Rm</th>
<th>Rd</th>
</tr>
</thead>
</table>

**T2**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>1 1 1 1 1</th>
<th>0 1 0</th>
<th>1 0 1</th>
<th>Rn</th>
<th>1 1 1</th>
<th>Rd</th>
<th>1 0 1 1</th>
<th>Rm</th>
</tr>
</thead>
</table>

**T2**

REVSH{<c>}{<q>} .W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

REVSH{<c>}{<q>} <Rd>, <Rm>

d = Uint(Rd); m = Uint(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;

**CONstrained UNPREDICTable behavior**

If m != n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: m = Uint(Rn);.
- The instruction executes with the additional decode: m = Uint(Rm);.
- The value in the destination register is UNKNOWN.

For more information about the CONstrained UNPREDICTable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8> = SignExtend(R[m]<7:0>, 24);
    result<7:0> = R[m]<15:8>;
    R[d] = result;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Return From Exception loads two consecutive memory locations using an address in a base register:

- The word loaded from the lower address is treated as an instruction address. The PE branches to it.
- The word loaded from the higher address is used to restore PSTATE. This word must be in the format of an SPSR.

An address adjusted by the size of the data loaded can optionally be written back to the base register.

The PE checks the value of the word loaded from the higher address for an illegal return event. See Illegal return events from AArch32 state.

RFE is UNDEFINED in Hyp mode and CONSTRAINED UNPREDICTABLE in User mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & P & U & 0 & W & 1 & Rn & (0)(0)(0)(0)(1)(1)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)
\end{array}
\]

Decrement After (P == 0 && U == 0)

RFE{<c>}{<q>}<Rn>{!} // (Preferred syntax)
RFE{FA}{<c>}{<q>}<Rn>{!} // (Alternate syntax, Full Ascending stack)

Decrement Before (P == 1 && U == 0)

RFE{DB}{<c>}{<q>}<Rn>{!} // (Preferred syntax)
RFE{EA}{<c>}{<q>}<Rn>{!} // (Alternate syntax, Empty Ascending stack)

Increment After (P == 0 && U == 1)

RFE{IA}{<c>}{<q>}<Rn>{!} // (Preferred syntax)
RFE{FD}{<c>}{<q>}<Rn>{!} // (Alternate syntax, Full Descending stack)

Increment Before (P == 1 && U == 1)

RFE{IB}{<c>}{<q>}<Rn>{!} // (Preferred syntax)
RFE{ED}{<c>}{<q>}<Rn>{!} // (Alternate syntax, Empty Descending stack)

n = UInt(Rn);
wback = (W == '1'); increment = (U == '1'); wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;

T1

\[
\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & W & 1 & Rn & (1)(1)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)
\end{array}
\]

T1

RFE{DB}{<c>}{<q>}<Rn>{!} // (Outside or last in IT block, preferred syntax)
RFE{FA}{<c>}{<q>}<Rn>{!} // (Outside or last in IT block, alternate syntax, Full Ascending stack)

n = UInt(Rn); wback = (W == '1'); increment = FALSE; wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
T2

RFE{{IA}{<c>{<q>}}{<q>}{!}} // (Outside or last in IT block, preferred syntax)
RFEFD{{<c>{<q>}}{<q>}{!}} // (Outside or last in IT block, alternate syntax, Full Descending stack)

n = UInt(Rn);  wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() & LastInITBlock() then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

IA  For encoding A1: is an optional suffix to indicate the Increment After variant.
    For encoding T2: is an optional suffix for the Increment After form.

<c>  For encoding A1: see Standard assembler syntax fields. <c> must be AL or omitted.
    For encoding T1 and T2: see Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<Rn>  Is the general-purpose base register, encoded in the “Rn” field.

!  The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

RFEFA, RFEEA, RFEFD, and RFEED are pseudo-instructions for RFEDA, RFEDB, RFEIA, and RFEIB respectively, referring to their use for popping data from Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.EL == EL0 then
        UNPREDICTABLE;          // UNDEFINED or NOP
    else
        address = if increment then R[n] else R[n]-8;
        if wordhigher then address = address+4;
        new_pc_value = MemA[address,4];
        spsr = MemA[address+4,4];
        if wback then R[n] = if increment then R[n]+8 else R[n]-8;
        AArch32.ExceptionReturn(new_pc_value, spsr);

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
ROR (immediate)

Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>Rd</td>
<td>!= 00000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td>S</td>
<td>imm5</td>
<td>stype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MOV, shift or rotate by value

ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

T3

| 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 1   | 1   | 0   | 1   | 0   | 0   | 1   | 0   | 0   | 1   | 1   | 1   | 1   | (0)  | imm3 | Rd  | imm2 | 1   | 1   | Rm  |
| S   | imm3 | stype |

MOV, shift or rotate by value ((imm3 == 000 && imm2 == 00))

ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

  For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

  For encoding T3: is the general-purpose source register, encoded in the "Rm" field.
- `<imm>` For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

  For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.
Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVs (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVs (register-shifted register).
- The description of MOV, MOVs (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

![A1 encoding diagram]

Not flag setting

ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

T1

![T1 encoding diagram]

Rotate right

ROR{<c>}{<q>} {<Rdm>,} <Rdm>, <Rs> // (Inside IT block)

is equivalent to

MOV{<c>}{<q>} <Rdm>, <Rdm>, ROR <Rs>

and is the preferred disassembly when InITBlock().

T2

![T2 encoding diagram]

Not flag setting

ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)

ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to
MOV\{<c>\}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

**Assembler Symbols**

- \(<c>\)
  See *Standard assembler syntax fields*.

- \(<q>\)
  See *Standard assembler syntax fields*.

- \(<Rdm>\)
  Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

- \(<Rd>\)
  Is the general-purpose destination register, encoded in the "Rd" field.

- \(<Rm>\)
  Is the first general-purpose source register, encoded in the "Rm" field.

- \(<Rs>\)
  Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation**

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
RORS (immediate)

Rotate Right, setting flags (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| != 1111                                       | != 00000                                      |
| Rd                                            | imm5                                         |
| cond                                          | stype                                        |

MOVS, shift or rotate by value

RORS{<c>}{<q>} {<Rd>}, <Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

T3

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 0000 &amp; imm2 == 00</td>
</tr>
<tr>
<td>Rd</td>
</tr>
<tr>
<td>imm2</td>
</tr>
<tr>
<td>stype</td>
</tr>
</tbody>
</table>

MOVS, shift or rotate by value ((imm3 == 000 && imm2 == 00))

RORS{<c>}{<q>} {<Rd>}, <Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

**Operation**

The description of **MOV, MOVS (register)** gives the operational pseudocode for this instruction.
RORS (register)

Rotate Right, setting flags (register) provides the value of the contents of a register rotated by a variable number of bits, and updates the condition flags based on the result. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

| 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 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| cond | != 1111 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | (0) | (0) | (0) | Rd | Rs | 0 | 1 | 1 | 1 | Rm | stype |
| Flag setting |
| RORS{<c>{<q>}}{<Rd>,} <Rm>, <Rs> |
| is equivalent to |
| MOV{<c>{<q>}} <Rd>, <Rm>, ROR <Rs> |
| and is always the preferred disassembly. |

T1

| bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| op | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | Rs | Rdm |
| Rotate right |
| RORS{<q>} {<Rdm>,} <Rdm>, <Rs> // (Outside IT block) |
| is equivalent to |
| MOV{<q>} <Rdm>, <Rdm>, ROR <Rs> |
| and is the preferred disassembly when !InITBlock(). |

T2

| bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| stype | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | Rm | 1 | 1 | 1 | 1 | Rd | 0 | 0 | 0 | 0 | Rs |
| Flag setting |
| RORS.W {<Rd>,} <Rm>, <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1) |
| RORS{<c>{<q>}}{<Rd>,} <Rm>, <Rs> |
| is equivalent to |
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rdm>` Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the first general-purpose source register, encoded in the "Rm" field.
- `<Rs>` Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation**

The description of **MOV, MOVS (register-shifted register)** gives the operational pseudocode for this instruction.
RRX

Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

This is an alias of MOV, MOVs (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVs (register).
- The description of MOV, MOVs (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| != 1111                      | 0 0 0 1 1 0 1 0 (0) (0) (0) Rd 0 0 0 0 0 1 1 0 Rm |

**MOV, rotate right with extend**

RRX{<c>}{<q>}{<Rd>,}<Rm>

is equivalent to

MOV{<c>}{<q>}{<Rd>,}<Rm>, RRX

and is always the preferred disassembly.

**T3**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 1 1 1 0 1 0 1 0 0 1 0 0 1 1 1 (0) 0 0 0 Rd 0 0 1 1 Rm |

**MOV, rotate right with extend**

RRX{<c>}{<q>}{<Rd>,}<Rm>

is equivalent to

MOV{<c>}{<q>}{<Rd>,}<Rm>, RRX

and is always the preferred disassembly.

**Assembler Symbols**

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
  
  For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
  
  For encoding T3: is the general-purpose source register, encoded in the "Rm" field.
Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
RRXS

Rotate Right with Extend, setting flags provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31]. If the destination register is not the PC, this instruction updates the condition flags based on the result, and bit[0] is shifted into the Carry flag.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| != 1111         | 0 0 0 1 1 0 1   | (0) (0) (0)     |
| cond            | Rd              |
| imm5            |
| stype           |

MOVS, rotate right with extend

RRXS{<c>}{<q>}{<Rd>,} <Rm>

is equivalent to

MOVS{<c>}{<q>}{<Rd>,} <Rm>, RRX

and is always the preferred disassembly.

T3

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 1 0 1 0 1   | 0 0 1 0 1 1 1   | (0) 0 0         |
| S               |
| imm3            |
| imm2 stype      |

MOVS, rotate right with extend

RRXS{<c>}{<q>}{<Rd>,} <Rm>

is equivalent to

MOVS{<c>}{<q>}{<Rd>,} <Rm>, RRX

and is always the preferred disassembly.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

**Operation**

The description of [MOV, MOVS (register)](MOV_MOVS_register) gives the operational pseudocode for this instruction.
RSB, RSBS (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register.
If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:
- The RSB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores \( PSTATE \) from \( \text{SPSR}_{\text{current\_mode}} \).
  - The PE checks \( \text{SPSR}_{\text{current\_mode}} \) for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (\( \text{A1} \)) and T32 (\( \text{T1} \) and \( \text{T2} \)).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 0 0 1 0 0 1 1 S | Rn | Rd | imm12 |

**cond**

**RSB (S == 0)**

\[
\text{RSB}\{<c>\}<q> \{<Rd>, \} <Rn>, \#<const>
\]

**RSBS (S == 1)**

\[
\text{RSBS}\{<c>\}<q> \{<Rd>, \} <Rn>, \#<const>
\]

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1'); \ \text{imm32} = A32ExpandImm(imm12);
\]

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Rn | Rd | |

**T1**

\[
\text{RSB}<c>\{<q> \{<Rd>, \} <Rn>, \#0} \ (\text{Inside IT block})
\]

\[
\text{RSBS}<q> \{<Rd>, \} <Rn>, \#0 \ (\text{Outside IT block})
\]

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = !\text{InITBlock}(); \ \text{imm32} = \text{Zeros}(32); \ // \text{immediate} = \#0
\]

**T2**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | i | 0 | 1 | 1 | 1 | 0 | S | Rn | 0 | imm3 | Rd | imm8 |
RSB \((S == 0)\)

\[\text{RSB}<c>.W \{<Rd>,} <Rn>, #0 \] // (Inside IT block)
\[\text{RSB}{<c>}\{<q>\} \{<Rd>,} <Rn>, #<\text{const}>\]

RSBS \((S == 1)\)

\[\text{RSBS}.W \{<Rd>,} <Rn>, #0 \] // (Outside IT block)
\[\text{RSBS}{<c>}\{<q>\} \{<Rd>,} <Rn>, #<\text{const}>\]

\[d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{setflags} = (S == '1'); \quad \text{imm32} = \text{T32ExpandImm}(i:imm3:imm8);\]
\[\text{if } d == 15 \quad || \quad n == 15 \quad \text{then}\] UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the \textit{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see \textit{Architectural Constraints on UNPREDICTABLE behaviors}.

**Assembler Symbols**

\(<c>\) See \textit{Standard assembler syntax fields}.

\(<q>\) See \textit{Standard assembler syntax fields}.

\(<Rd>\) For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<Rn>\). Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see \textit{Pseudocode description of operations on the AArch32 general-purpose registers and the PC}.
- For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_\textit{<current_mode>}. For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<Rn>\).

\(<Rn>\) For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the general-purpose source register, encoded in the "Rn" field.

\(<\text{const}>\) For encoding A1: an immediate value. See \textit{Modified immediate constants in A32 instructions} for the range of values.

For encoding T2: an immediate value. See \textit{Modified immediate constants in T32 instructions} for the range of values.

**Operation**

\[\text{if } \text{ConditionPassed()} \text{ then}\]
\[\text{EncodingSpecificOperations();}\]
\[(result, nzc) = \text{AddWithCarry}(\text{NOT}(R[n]), \text{imm32}, '1');\]
\[\text{if } d == 15 \text{ then } \quad \text{// Can only occur for A32 encoding}\]
\[\text{if setflags then}\]
\[\text{ALUExceptionReturn}(result);\]
\[\text{else}\]
\[\text{ALUWritePC}(result);\]
\[\text{else}\]
\[R[d] = result;\]
\[\text{if setflags then}\]
\[\text{PSTATE.<N,Z,C,V> = nzc;}\]

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 0 0 0 0 1 1 S</td>
</tr>
</tbody>
</table>

**cond**

#### RSB, rotate right with extend (S == 0 & imm5 == 00000 & stype == 11)

```c
RSB{<c>{<q>}}{<Rd>,} <Rn>, <Rm>, RRX
```

#### RSB, shift or rotate by value (S == 0 & !(imm5 == 00000 & stype == 11))

```c
RSB{<c>{<q>}}{<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### RSBS, rotate right with extend (S == 1 & imm5 == 00000 & stype == 11)

```c
RSBS{<c>{<q>}}{<Rd>,} <Rn>, <Rm>, RRX
```

#### RSBS, shift or rotate by value (S == 1 & !(imm5 == 00000 & stype == 11))

```c
RSBS{<c>{<q>}}{<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

```c
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 1 1 1 0 S</td>
</tr>
</tbody>
</table>
RSB, rotate right with extend \((S == 0 \&\& \text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11)\)

\[
\text{RSB}\{<c>\}{<q>\} \{<Rd>,\} \{<Rn>, \} \{<Rm>, \} \text{, RRX}
\]

RSB, shift or rotate by value \((S == 0 \&\& !((\text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11))\)

\[
\text{RSB}\{<c>\}{<q>\} \{<Rd>,\} \{<Rn>, \} \{<Rm> \{, <shift> \#<amount>\} \}
\]

RSBS, rotate right with extend \((S == 1 \&\& \text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11)\)

\[
\text{RSBS}\{<c>\}{<q>\} \{<Rd>,\} \{<Rn>, \} \{<Rm>, \} \text{, RRX}
\]

RSBS, shift or rotate by value \((S == 1 \&\& !((\text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11))\)

\[
\text{RSBS}\{<c>\}{<q>\} \{<Rd>,\} \{<Rn>, \} \{<Rm> \{, <shift> \#<amount>\} \}
\]

\[
d = \text{UInt}(Rd); \ \ n = \text{UInt}(Rn); \ \ m = \text{UInt}(Rm); \ \ \text{setflags} = (S == '1');
\]
\[
\text{(shift}_t, \ \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \ \text{imm3}:\text{imm2});
\]
\[
\text{if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

For more information about the \text{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see \textit{Architectural Constraints on UNPREDICTABLE behaviors}.

\textbf{Assembler Symbols}

\textbf{<c> See Standard assembler syntax fields.}

\textbf{<q> See Standard assembler syntax fields.}

\textbf{<Rd>}

For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see \textit{Pseudocode description of operations on the AArch32 general-purpose registers and the PC}.
- For the RSBS variant, the instruction performs an exception return, that restores \text{PSTATE} from SPSR\_<current\_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

\textbf{<Rn>}

For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

\textbf{<Rm>}

For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

\textbf{<shift>}

Is the type of shift to be applied to the second source register, encoded in "stype":

\[
\begin{array}{c|c}
\text{stype} & \text{<shift>} \\
\hline
00 & \text{LSL} \\
01 & \text{LSR} \\
10 & \text{ASR} \\
11 & \text{ROR} \\
\end{array}
\]

\textbf{<amount>}

For encoding A1: is the shift amount, in the range 1 to 31 (when \text{<shift>} = \text{LSL} or \text{ROR}) or 1 to 32 (when \text{<shift>} = \text{LSR} or \text{ASR}) encoded in the "imm5" field as \text{<amount>} modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when \text{<shift>} = \text{LSL} or \text{ROR}) or 1 to 32 (when \text{<shift>} = \text{LSR} or \text{ASR}), encoded in the "imm3:imm2" field as \text{<amount>} modulo 32.
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, '1');
    if d == 15 then  // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Reverse Subtract (register-shifted register) subtracts a register value from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rs</td>
<td>0</td>
<td>stype</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag setting (S == 1)

RSBS{<c>}{<q>}{<Rd>,} {<Rn>, <Rm>, <shift> <Rs} 

Not flag setting (S == 0)

RSB{<c>}{<q>}{<Rd>,} {<Rn>, <Rm>, <shift> <Rs} 

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

\[
\text{setflags} = (S == '1'); \quad \text{shift_t} = \text{DecodeRegShift}(\text{stype});
\]

\[
\text{if } d == 15 || n == 15 || m == 15 || s == 15 \text{ then UNPREDICTABLE;}
\]

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<\text{c}> See Standard assembler syntax fields.

<\text{q}> See Standard assembler syntax fields.

<\text{Rd}> Is the general-purpose destination register, encoded in the "Rd" field.

<\text{Rn}> Is the first general-purpose source register, encoded in the "Rn" field.

<\text{Rm}> Is the second general-purpose source register, encoded in the "Rm" field.

<\text{shift}> Is the type of shift to be applied to the second source register, encoded in "stype":

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

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

\[
\text{if } \text{ConditionPassed}() \text{ then }
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{shift_n} = \text{UInt}(R[s] \text{<7:0>});
\]

\[
\text{shifted} = \text{Shift}(R[m], \text{shift_t}, \text{shift_n}, \PSTATE.C);
\]

\[
\text{(result, nzcv)} = \text{AddWithCarry}(\text{NOT}(R[n]), \text{shifted}, '\text{1}');
\]

\[
R[d] = \text{result};
\]

\[
\text{if setflags then}
\]

\[
\PSTATE.\text{<N,Z,C,V>} = \text{nzcv};
\]

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
RSC, RSCS (immediate)

Reverse Subtract with Carry (immediate) subtracts a register value and the value of NOT (Carry flag) from an immediate value, and writes the result to the destination register. If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

**A1**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>imm12</td>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RSC (S == 0)**

RSC{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**RSCS (S == 1)**

RSCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

\[d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{setflags} = (S == '1'); \quad \text{imm32} = \text{A32ExpandImm}(\text{imm12})\]

**Assembler Symbols**

- **<c>**
  - See *Standard assembler syntax fields*.

- **<q>**
  - See *Standard assembler syntax fields*.

- **<Rd>**
  - The general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. ARM deprecates using the PC as the destination register, but if the PC is used:
    - For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
    - For the RSCS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

- **<Rn>**
  - The general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

- **<const>**
  - An immediate value. See *Modified immediate constants in A32 instructions* for the range of values.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(NOT(R[n]), imm32, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Reverse Subtract with Carry (register) subtracts a register value and the value of NOT (Carry flag) from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------------------|---|---|---|---|
| != 1111 | 0 0 0 0 | 1 1 1 | S | Rn | Rd | imm5 | stype | 0 | Rm |
```

cond

RSC, rotate right with extend (S == 0 & imm5 == 00000 & stype == 11)

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSC, shift or rotate by value (S == 0 & !(imm5 == 00000 & stype == 11))

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

RSCS, rotate right with extend (S == 1 & imm5 == 00000 & stype == 11)

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSCS, shift or rotate by value (S == 1 & !(imm5 == 00000 & stype == 11))

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. ARM deprecates using the PC as the destination register, but if the PC is used:
  - For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  - For the RSCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
<shift>  Is the type of shift to be applied to the second source register, encoded in “stype”:

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

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
  (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
  if d == 15 then
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
  else
    R[d] = result;
    if setflags then
      PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Reverse Subtract (register-shifted register) subtracts a register value and the value of NOT (Carry flag) from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

<table>
<thead>
<tr>
<th>!</th>
<th>1111</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Rs</th>
<th>0</th>
<th>stype</th>
<th>1</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag setting (S == 1)

RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

Not flag setting (S == 0)

RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<
See Standard assembler syntax fields.
>
See Standard assembler syntax fields.
<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>
Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>
Is the second general-purpose source register, encoded in the "Rm" field.
<shift>
Is the type of shift to be applied to the second source register, encoded in "stype":

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

<Rs>
Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets PSTATE GE according to the results of the additions. It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | Rn | Rd | (1) | (1) | (1) | (1) | 0 | 0 | 0 | 1 | Rm |

cond

A1

SADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 0 | 0 | 0 | Rm |

T1

SADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0> = sum1<15:0>;
    R[d]<31:16> = sum2<15:0>;
    PSTATE.GE<1:0> = if sum1 >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if sum2 >= 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111| 0  | 1  | 1  | 0  | 0  | 0  | 1  | Rn | Rd | (1)|(1)|(1)|(1)| 1 | 0 | 0 | 1 | Rm |

cond

A1

SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | 1  | 1  | 1  | 1 | 0 | 1 | 0 | 0 | 0 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 0 | 0 | 0 | Rm |

T1

SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0> = sum1<7:0>;
    R[d]<15:8> = sum2<7:0>;
    R[d]<23:16> = sum3<7:0>;
    R[d]<31:24> = sum4<7:0>;
    PSTATE.GE<0> = if sum1 >= 0 then '1' else '0';
    PSTATE.GE<1> = if sum2 >= 0 then '1' else '0';
    PSTATE.GE<2> = if sum3 >= 0 then '1' else '0';
    PSTATE.GE<3> = if sum4 >= 0 then '1' else '0';

SADD8
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets $\text{PSTATE.GE}$ according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
SEND 0 1 1 0 0 0 1 Rn Rd (1)(1)(1)(1) 0 0 1 1 Rm
! = 1111 0 1 1 0 0 0 1
cond
```

A1

SASX{<c>}{<q>} {<Rd>,} {<Rn>,} {<Rm>}

d = Uint(Rd);  n = Uint(Rn);  m = Uint(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

```
SEND 1 1 1 1 0 1 0 0 0 0 0 1 Rn 1 1 1 1 Rd 0 0 0 0 Rm
```

T1

SASX{<c>}{<q>} {<Rd>,} {<Rn>,} {<Rm>}

d = Uint(Rd);  n = Uint(Rn);  m = Uint(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

Operation

If $\text{ConditionPassed()}$ then

- EncodingSpecificOperations();
- diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
- sum = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
- R[d]<15:0> = diff<15:0>;
- R[d]<31:16> = sum<15:0>;
- PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
- PSTATE.GE<3:2> = if sum >= 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Speculation Barrier is a barrier that controls speculation.
The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that
appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels
  as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has
  not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative
allocation into any caching structure where the allocation of that entry could be indicative of any data value present in
memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has
  not generated an exception. The potentially exception generating instruction can complete once it is known
  not to be speculative, and all data values generated by instructions appearing in program order before the SB
  instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the
speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB
instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

```
SB{<q>}

// No additional decoding required
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 1 1 0 1 1 (1)(1)(1)(1)(1)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)
```

T1

```
SB{<q>}

if InitBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints
on UNPREDICTABLE behaviors.

Assembler Symbols

```
<q> See Standard assembler syntax fields.
```

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  SpeculationBarrier();
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3; Build timestamp: 2020-09-30T21:35
SBC, SBCS (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-----------------|----------------|
| ! = 1111 | 0 0 1 0 1 1 0 | S | Rd | imm12 |

cond

SBC (S == 0)

SBC{<c}>{<q>} {<Rd>,} <Rn>, #<const>

SBCS (S == 1)

SBCS{<c}>{<q>} {<Rd>,} <Rn>, #<const>

d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = A32ExpandImm(imm12);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-----------------|----------------|
| 1 1 1 1 0 | i | 0 1 0 1 1 | S | Rd | imm8 |

SBC (S == 0)

SBC{<c}>{<q>} {<Rd>,} <Rn>, #<const>

SBCS (S == 1)

SBCS{<c}>{<q>} {<Rd>,} <Rn>, #<const>

d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- <c> See Standard assembler syntax fields.
- <q> See Standard assembler syntax fields.
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const>
For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for ⟨Rd⟩ identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR <current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

cond

SBC, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

SBC{<c>{<q>}} {<Rd>,} <Rn>, <Rm>, RRX

SBC, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

SBC{<c>{<q>}} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

SBCS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

SBCS{<c>{<q>}} {<Rd>,} <Rn>, <Rm>, RRX

SBCS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

SBCS{<c>{<q>}} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S == '1');
\]

\[(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5});\]

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

T1

SBC{<c>{<q>}} {<Rd>,} <Rn>, <Rm> // (Inside IT block)

SBCS{<c>{<q>}} {<Rd>,} <Rn>, <Rm> // (Outside IT block)

\[
d = \text{UInt}(Rdn); \quad n = \text{UInt}(Rdn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = !\text{InITBlock}();
\]

\[(\text{shift}_t, \text{shift}_n) = (\text{SRTypelSL}, 0);\]

T2
SBC, rotate right with extend (S == 0 &&imm3 == 000 && imm2 == 00 && stype == 11)

SBC{<c>{<q>}}{<Rd>,} <Rn>, <Rm>, RRX

SBC, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SBC{<c>}{<q>}{<Rd>,} <Rn>, <Rm} // (Inside IT block, and <Rd>, <Rn>, <Rm} can be represented in T1)
SBC{<c>{{<q>}}{<Rd>,} <Rn}, <Rm} {, <shift} #<amount} }

SBCS, rotate right with extend (S == 1 &&imm3 == 000 && imm2 == 00 && stype == 11)

SBCS{<c>{{<q>}}{<Rd>,} <Rn>, <Rm}, RRX

SBCS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SBCS.W {<Rd>,} <Rn>, <Rm} // (Outside IT block, and <Rd>, <Rn}, <Rm} can be represented in T1)
SBCS{<c>{{<q>}}{<Rd>,} <Rn}, <Rm} {, <shift} #<amount} }

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.
<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.
<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
  • For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  • For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in "stype":

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

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

SBC, SBCS (register)
For encoding T2: is the shift amount, in the range 1 to 31 (when \(<\text{shift}> = \text{LSL or ROR}\)) or 1 to 32 (when \(<\text{shift}> = \text{LSR or ASR}\)), encoded in the “imm3:imm2” field as \(<\text{amount}>\) modulo 32.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SBC, SBCS (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

\[
\begin{array}{cccccccccccccccccc}
\text{cond} & ! & = & 1 & 1 & 1 & 1 & \text{S} & \text{Rn} & \text{Rd} & \text{Rs} & 0 & \text{stype} & 1 & \text{Rm} \\
\end{array}
\]

Flag setting (S == 1)

SBC{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}, <shift> <Rs>

Not flag setting (S == 0)

SBC{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}, <shift> <Rs>

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

setflags = (S == '1'); \ shift_t = \text{DecodeRegShift}(\text{stype});

if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- <c> See Standard assembler syntax fields.
- <q> See Standard assembler syntax fields.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the type of shift to be applied to the second source register, encoded in "stype":

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

- <Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | widthm1 | Rd | lsb | 1 | 0 | 1 | Rn |
| cond |

A1

```python
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

d = UInt(Rd);  n = UInt(Rn);
lbbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 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 | 1 | 0 | 0 | Rn | 0 | imm3 | Rd | imm2 | (0) | widthm1 |

T1

```python
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

d = UInt(Rd);  n = UInt(Rn);
lbbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

<
See Standard assembler syntax fields.
<
See Standard assembler syntax fields.
<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>
Is the general-purpose source register, encoded in the "Rn" field.
<lsb>
For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.
For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.
<width>
Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

**Operation**

```python
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```
CONSTRAINED UNPREDICTABLE behavior

If $\text{msbit} > 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition flags are not affected.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{ccccccccccccccccccccccccccc}
\hline
!= & 1111 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \text{Rd} & (1)(1)(1)(1) & \text{Rm} & 0 & 0 & 0 & 1 & \text{Rn} \\
\end{array}
\]

cond Ra

A1

\[
\text{SDIV}\{<c>\}{<q>} \{<Rd>,} <\text{Rn}, <\text{Rm}>
\]

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

if d == 15 \&\& n == 15 \&\& m == 15 \&\& a != 15 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

T1

\[
\begin{array}{ccccccccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & \text{Rn} & (1)(1)(1)(1) & \text{Rd} & 1 & 1 & 1 & 1 & \text{Rm} \\
\end{array}
\]

Ra

T1

\[
\text{SDIV}\{<c>\}{<q>} \{<Rd>,} <\text{Rn}, <\text{Rm}>
\]

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

if d == 15 \&\& n == 15 \&\& m == 15 \&\& a != 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRANGED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- \(<c>\) See Standard assembler syntax fields.
- \(<q>\) See Standard assembler syntax fields.
- \(<\text{Rd}>\) Is the general-purpose destination register, encoded in the "Rd" field.
<Rn> is the first general-purpose source register holding the dividend, encoded in the "Rn" field.

<Rm> is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Overflow
If the signed integer division 0x80000000 / 0xFFFFFFFF is performed, the pseudocode produces the intermediate integer result $2^{31}$, that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to <Rd> must be the bottom 32 bits of the binary representation of $2^{31}$. So the result of the division is 0x80000000.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    if SInt(R[m]) == 0 then
        result = 0;
    else
        result = RoundTowardsZero(Real(SInt(R[n])) / Real(SInt(R[m])));
    R[d] = result<31:0>;
```

Internal version only: isa v01.19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the PSTATE.GE flags.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Condition**

SEL{<c>{<q>}}{<Rd>,} <Rn>, <Rm>

d = Uint(Rd); n = Uint(Rn); m = Uint(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1| Rm |

**Condition**

SEL{<c>{<q>}}{<Rd>,} <Rn>, <Rm>

d = Uint(Rd); n = Uint(Rn); m = Uint(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

if ConditionPassed() then

    EncodingSpecificOperations();

    if PSTATE.GE<0> == '1' then R[n]<7:0> else R[m]<7:0>;
    if PSTATE.GE<1> == '1' then R[n]<15:8> else R[m]<15:8>;
    if PSTATE.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
    if PSTATE.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SETEND

Set Endianness writes a new value to `PSTATE.E`.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

`SETEND{<q>} <endian_specifier> // (Cannot be conditional)`

```plaintext
set_bigend = (E == '1');
```

T1

`SETEND{<q>} <endian_specifier> // (Not permitted in IT block)`

```plaintext
set_bigend = (E == '1');
if InITBlock() then UNPREDICTABLE;
```

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

*<q>*  
See *Standard assembler syntax fields*.

*<endian_specifier>*  
Is the endianness to be selected, and the value to be set in PSTATE.E, encoded in "E":

<table>
<thead>
<tr>
<th>E</th>
<th>&lt;endian_specifier&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LE</td>
</tr>
<tr>
<td>1</td>
<td>BE</td>
</tr>
</tbody>
</table>

Operation

EncodingSpecificOperations();
AArch32.CheckSETENDEnabled();
PSTATE.E = if set_bigend then '1' else '0';
SETPAN

Set Privileged Access Never writes a new value to `PSTATE.PAN`. This instruction is available only in privileged mode and it is a NOP when executed in User mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

(Armv8.1)

```
1 1 1 1 0 0 0 1 0 0 0 1 0 1 (0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)
```

T1

(Armv8.1)

```
1 0 1 1 0 1 1 0 0 0 0 1 (1)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)
```

Assembler Symbols

- `<q>`: See *Standard assembler syntax fields*.
- `<imm>`: Is the unsigned immediate 0 or 1, encoded in the "imm1" field.

Operation

```
EncodingSpecificOperations();
if PSTATE.EL != EL0 then
PSTATE.PAN = value;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 and Send Event*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 0 1 1 0 0 0 0 0 1 1 1 0</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>
```

A1

`SEV{<c>}{<q>}`

// No additional decoding required

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 1 1 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
```

T1

`SEV{<c>}{<q>}`

// No additional decoding required

T2

```
<table>
<thead>
<tr>
<th>15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0 0 1 1 1 0 1 1 1 0</td>
</tr>
</tbody>
</table>
```

T2

`SEV{<c>}.W`

// No additional decoding required

For more information about the constrained unpredictable behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>`  
  See *Standard assembler syntax fields*.
- `<q>`  
  See *Standard assembler syntax fields*.

**Operation**

```java
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

![A1 encoding](image)

```plaintext
A1
SEVL{<c>}{<q>}
// No additional decoding required
```

### T1

![T1 encoding](image)

```plaintext
T1
SEVL{<c>}{<q>}
// No additional decoding required
```

### T2

![T2 encoding](image)

```plaintext
T2
SEVL{<c>}.W
// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.

### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEventLocal();
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| != 1111 | 0 1 1 0 0 0 1 1 | Rn | Rd | (1) | (1) | (1) | (1) | 0 | 0 | 0 | 1 | Rm |

SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

\[\text{d} = \text{UInt}(\text{Rd}); \text{ n} = \text{UInt}(\text{Rn}); \text{ m} = \text{UInt}(\text{Rm});\]
if \(d == 15 \|| \ n == 15 \|| \ m == 15\) then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 0 0 0 0 1 1 1 1</td>
</tr>
</tbody>
</table>

SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

\[\text{d} = \text{UInt}(\text{Rd}); \text{ n} = \text{UInt}(\text{Rn}); \text{ m} = \text{UInt}(\text{Rm});\]
if \(d == 15 \|| \ n == 15 \|| \ m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    \[\text{sum1} = \text{SInt}(\text{R}[n]<15:0>) + \text{SInt}(\text{R}[m]<15:0>);\]
    \[\text{sum2} = \text{SInt}(\text{R}[n]<31:16>) + \text{SInt}(\text{R}[m]<31:16>);\]
    \[\text{R}[d]<15:0> = \text{sum1}<16:1>;\]
    \[\text{R}[d]<31:16> = \text{sum2}<16:1>;\]

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

A1

SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 0 1 0 0</td>
</tr>
<tr>
<td>T1</td>
</tr>
</tbody>
</table>

T1

SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then

    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0> = sum1<8:1>;
    R[d]<15:8> = sum2<8:1>;
    R[d]<23:16> = sum3<8:1>;
    R[d]<31:24> = sum4<8:1>;
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 0 1 0 1 0 1 0 Rn 1 1 1 1 Rd 0 0 1 0 Rm
```

```assembly
SHASX{<c>}{<q>}{<Rd>,}<Rn>,<Rm>
```

- $d = \text{UInt}(Rd)$; $n = \text{UInt}(Rn)$; $m = \text{UInt}(Rm)$;
- if $d == 15 || n == 15 || m == 15$ then UNPREDICTABLE;

**T1**

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 0 1 0 1 0 1 0 Rn 1 1 1 1 Rd 0 0 1 0 Rm
```

```assembly
SHASX{<c>}{<q>}{<Rd>,}<Rn>,<Rm>
```

- $d = \text{UInt}(Rd)$; $n = \text{UInt}(Rn)$; $m = \text{UInt}(Rm)$;
- if $d == 15 || n == 15 || m == 15$ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

If `ConditionPassed()` then

```
EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0> = diff<16:1>;
    R[d]<31:16> = sum<16:1>;
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 1 1 0 0 0 1 1 Rn</td>
</tr>
</tbody>
</table>

cond

#### A1

SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

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

if \( d == 15 \land n == 15 \land m == 15 \) then UNPREDICTABLE;

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 0 1 1 1 1 1 0 Rn</td>
</tr>
</tbody>
</table>

#### T1

SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

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

if \( d == 15 \land n == 15 \land m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

#### Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    sum = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0> = sum<16:1>;
    R[d]<31:16> = diff<16:1>;
```

#### Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111| 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | Rn | Rd | (1)| (1) | (1) | (1) | 0  | 1  | 1  | 1  | Rm | cond

**A1**

\`
\text{SHSUB16}\{<c>\}\{<q>\} \{<Rd>,}\ <Rn>, <Rm>
\`

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

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**T1**

\`
\text{SHSUB16}\{<c>\}\{<q>\} \{<Rd>,}\ <Rn>, <Rm>
\`

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

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

For more information about the constrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

### Operation

If `ConditionPassed()` then

\[
\begin{align*}
\text{EncodingSpecificOperations();} \\
diff1 &= \text{SInt}(R[n]<15:0>) - \text{SInt}(R[m]<15:0>); \\
diff2 &= \text{SInt}(R[n]<31:16>) - \text{SInt}(R[m]<31:16>); \\
R[d]<15:0> &= \text{diff1}<16:1>; \\
R[d]<31:16> &= \text{diff2}<16:1>;
\end{align*}
\]

### Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to
the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

A1

SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

T1

SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.
<Rd> Is the general-purpose destination register, encoded in the "Rd" field.
<Rn> Is the first general-purpose source register, encoded in the "Rn" field.
<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0> = diff1<8:1>;
    R[d]<15:8> = diff2<8:1>;
    R[d]<23:16> = diff3<8:1>;
    R[d]<31:24> = diff4<8:1>;
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

- The response of this instruction to 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 a Secure Monitor Call exception. For more information see Secure Monitor Call (SMC) exception.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in User mode.

If the values of HCR.TSC and SCR.SCD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception that is taken to EL3. When EL3 is using AArch32 this exception is taken to Monitor mode. When EL3 is using AArch64, it is the SCR_EL3.SMD bit, rather than the SCR.SCD bit, that can change the effect of executing an SMC instruction.

If the value of HCR.TSC is 1, execution of an SMC instruction in a Non-secure EL1 mode generates an exception that is taken to EL2, regardless of the value of SCR.SCD. When EL2 is using AArch32, this is a Hyp Trap exception that is taken to Hyp mode. For more information see Traps to Hyp mode of Non-secure EL1 execution of SMC instructions.

If the value of HCR.TSC is 0 and the value of SCR.SCD is 1, the SMC instruction is:

- UNDEFINED in Non-secure state.
- CONSTRAINED UNPREDICTABLE if executed in Secure state at EL1 or higher.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
! = 1111 0 0 0 1 0 1 1 0 (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) 0 1 1 1 imm4
cond
```

If InITBlock() && !LastInITBlock() then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 1 1 1 1 1 1 1 1 imm4 1 0 0 (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0) (0)
```

if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<imm4>` Is a 4-bit unsigned immediate value, in the range 0 to 15, encoded in the “imm4” field. This is ignored by the PE. The Secure Monitor Call exception handler (Secure Monitor code) can use this value to determine what service is being requested, but Arm does not recommend this.
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32.CheckForSMCUndefOrTrap();
    if !ELUsingAArch32(EL3) then
        if SCR_EL3.SMD == '1' then
            // SMC disabled.
            UNDEFINED;
        else
            if SCR.SCD == '1' then
                // SMC disabled
                if IsSecure() then
                    // Executes either as a NOP or UNALLOCATED.
                    c = ConstrainUnpredictable(Unpredictable_SMD);
                    assert c IN {Constraint_NOP, Constraint_UNDEF};
                    if c == Constraint_NOP then EndOfInstruction();
                    UNDEFINED;
                
            end
        end
    else
        AArch64.CallSecureMonitor(Zeros(16));
    else
        AArch32.TakeSMCException();
    end
end

CONSTRAINED UNPREDICTABLE behavior

If SCR.SCD == '1' && IsSecure(), then one of the following behaviors must occur:

  • The instruction is UNDEFINED.
  • The instruction executes as NOP.
SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets PSTATE.Q to 1. It is not possible for overflow to occur during the multiplication.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| != 1111          | 0 0 0 1 0 0 0    | Rd               | Ra               | Rm               | 1 | M | N | 0 | Rn |
cond

SMLABB (M == 0 && N == 0)

SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLABT (M == 1 && N == 0)

SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLATB (M == 0 && N == 1)

SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLATT (M == 1 && N == 1)

SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

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

\[n\text{ high} = (N == '1'); \quad m\text{ high} = (M == '1');\]

if \(d == 15|| n == 15 || m == 15 || a == 15\) then UNPREDICTABLE;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 1 1 1 1 0 1 1 0 0 0 1 | Rn               | != 1111          | Rd               | 0 0 | N | M | Rm |

Rd

Rn
SMLABB \( N == 0 \&\& M == 0 \)

\[
\text{SMLABB}\{<c>\}\{<q>\} \quad <Rd>, <Rn>, <Rm>, <Ra>
\]

SMLABT \( N == 0 \&\& M == 1 \)

\[
\text{SMLABT}\{<c>\}\{<q>\} \quad <Rd>, <Rn>, <Rm>, <Ra>
\]

SMLATB \( N == 1 \&\& M == 0 \)

\[
\text{SMLATB}\{<c>\}\{<q>\} \quad <Rd>, <Rn>, <Rm>, <Ra>
\]

SMLATT \( N == 1 \&\& M == 1 \)

\[
\text{SMLATT}\{<c>\}\{<q>\} \quad <Rd>, <Rn>, <Rm>, <Ra>
\]

if Ra == '1111' then SEE "SMULBB, SMULBT, SMULTB, SMULTT";

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

\[
n_{\text{high}} = (N == '1'); \quad m_{\text{high}} = (M == '1');
\]

\[
\text{if } d == 15 \text{ || } n == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\) Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by \(<x>\)), encoded in the "Rn" field.

\(<Rm>\) Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by \(<y>\)), encoded in the "Rm" field.

\(<Ra>\) Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

if ConditionPassed() then

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{operand1} = \text{if } n_{\text{high}} \text{ then } R[n]<31:16> \text{ else } R[n]<15:0>;;
\]

\[
\text{operand2} = \text{if } m_{\text{high}} \text{ then } R[m]<31:16> \text{ else } R[m]<15:0>;;
\]

\[
\text{result} = \text{SInt}(\text{operand1}) * \text{SInt}(\text{operand2}) + \text{SInt}(R[a]);
\]

\[
R[d] = \text{result}<31:0>;;
\]

if result != SInt(result<31:0>) then  // Signed overflow

PSTATE.Q = '1';

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMLAD, SMLADX

Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications.

It has encodings from the following instruction sets: A32 ( A1 ) and T32 ( T1 ) .

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------|-------------------|-------------------|
| != 1111 | 0 1 1 1 0 0 0 0 | Rd | != 1111 | Rm | 0 0 M 1 | Rn |

cond | Ra

SMLAD (M == 0)

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLADXM (M == 1)

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

if Ra == '1111' then SEE "SMUAD";

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|
| 1 1 1 1 1 0 1 1 0 0 1 0 | Rn | != 1111 | Rd | 0 0 0 M | Rm |

Ra

SMLAD (M == 0)

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLADXM (M == 1)

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

if Ra == '1111' then SEE "SMUAD";

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<
See Standard assembler syntax fields.

<
See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.
<Rm> Is the second general-purpose source register, encoded in the "Rm" field.
<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Operation**

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```
SMLAL, SMLALS

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>0 0 0 0 1 1 1 S RdHi RdLo Rm 1 0 0 1 Rn</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

Flag setting (S == 1)

SMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Not flag setting (S == 0)

SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

\[
\begin{align*}
\text{dLo} &= \text{UInt}(\text{RdLo}); \\
\text{dHi} &= \text{UInt}(\text{RdHi}); \\
\text{n} &= \text{UInt}(\text{Rn}); \\
\text{m} &= \text{UInt}(\text{Rm}); \\
\text{setflags} &= (S == '1'); \\
\text{if} \ dLo == 15 || \ dHi == 15 || \ n == 15 || \ m == 15 \ \text{then UNPREDICTABLE;}
\end{align*}
\]

// Armv8-A removes UNPREDICTABLE for R13

CONstrained UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 1 1 1 0 0 Rn RdLo RdHi 0 0 0 0 Rm</td>
</tr>
</tbody>
</table>

T1

SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

\[
\begin{align*}
\text{dLo} &= \text{UInt}(\text{RdLo}); \\
\text{dHi} &= \text{UInt}(\text{RdHi}); \\
\text{n} &= \text{UInt}(\text{Rn}); \\
\text{m} &= \text{UInt}(\text{Rm}); \\
\text{setflags} &= \text{FALSE}; \\
\text{if} \ dLo == 15 || \ dHi == 15 || \ n == 15 || \ m == 15 \ \text{then UNPREDICTABLE;}
\end{align*}
\]

// Armv8-A removes UNPREDICTABLE for R13

if dHi == dLo then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<RdLo> Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
    // PSTATE.C, PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo $2^{64}$.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | != 1111 | 0 0 0 1 0 1 0 0 | RdHi | RdLo | Rm | 1 | M | N | 0 | Rn |
|cond|

**SMLALBB (M == 0 && N == 0)**

SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

**SMLALBT (M == 1 && N == 0)**

SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

**SMLALTB (M == 0 && N == 1)**

SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

**SMLALTT (M == 1 && N == 1)**

SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

\[dLo = \text{UInt}(RdLo); \quad dHi = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);\]
\[n\text{ high} = (N == '1'); \quad m\text{ high} = (M == '1');\]
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;

**CONstrained UNPREDICTABLE behavior**

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 1 0 1 1 1 1 0 0 | Rn | RdLo | RdHi | 1 0 | N | M | Rm |
SMLALBB (N == 0 && M == 0)
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALBT (N == 0 && M == 1)
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTB (N == 1 && M == 0)
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTT (N == 1 && M == 1)
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

\[
\begin{align*}
\text{dLo} &= \text{UInt}(RdLo); \quad \text{dHi} = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \\
\text{n_high} &= (N == '1'); \quad \text{m_high} = (M == '1'); \\
\text{if} \ dLo == 15 || \ dHi == 15 || \ n == 15 || \ m == 15 \ \text{then UNPREDICTABLE} \\
// \text{Armv8-A removes UNPREDICTABLE for R13} \\
\text{if} \ dHi == dLo \ \text{then UNPREDICTABLE}
\end{align*}
\]

**CONSTRANGED UNPREDICTABLE behavior**

If \( dHi == dLo \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<RdLo>` Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
- `<RdHi>` Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
- `<Rn>` For encoding A1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by `<x>`), encoded in the "Rn" field.

For encoding T1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by `<y>`), encoded in the "Rn" field.

- `<Rm>` For encoding A1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the "Rm" field.

For encoding T1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<x>`), encoded in the "Rm" field.

**Operation**

\[
\begin{align*}
\text{if } \text{ConditionPassed}() \ \text{then}
\quad \text{EncodingSpecificOperations();} \\
\quad \text{operand1} = \text{if } n \ \text{high then } R[n]<31:16> \ \text{else } R[n]<15:0>; \\
\quad \text{operand2} = \text{if } m \ \text{high then } R[m]<31:16> \ \text{else } R[m]<15:0>; \\
\quad \text{result} = SInt(\text{operand1}) * SInt(\text{operand2}) + SInt(R[dHi]:R[dLo]); \\
\quad R[dHi] = \text{result}<63:32>; \\
\quad R[dLo] = \text{result}<31:0>;
\end{align*}
\]
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Multiply Accumulate Long Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo $2^{64}$.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

| A1 | 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|    | != 1111 0 1 1 1 0 1 0 0 | RdHi | RdLo | Rm | 0 0 | M 1 | Rn |

cond

SMLALD (M == 0)

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALDX (M == 1)

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

| T1 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|    | 1 1 1 1 1 0 1 1 1 1 0 0 | Rn | RdLo | RdHi | 1 1 0 | M | Rm |

SMLALD (M == 0)

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALDX (M == 1)

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>`  
  See *Standard assembler syntax fields*.

- `<q>`  
  See *Standard assembler syntax fields*.

- `<RdLo>`  
  Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

- `<RdHi>`  
  Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

- `<Rn>`  
  Is the first general-purpose source register, encoded in the "Rn" field.

- `<Rm>`  
  Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets \textit{PSTATE}.Q to 1. No overflow can occur during the multiplication.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

\textbf{A1}

\begin{verbatim}

1 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\textbf{(!= 1111 | 0 0 0 1 0 0 1 0 | Rd | Ra | Rm | 1 | M | 0 | 0 | Rn)} \textbf{cond}
\end{verbatim}

\textbf{SMLAWB (M == 0)}

\begin{verbatim}
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
\end{verbatim}

\textbf{SMLAWT (M == 1)}

\begin{verbatim}
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
\end{verbatim}

\begin{verbatim}
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
\end{verbatim}

\textbf{T1}

\begin{verbatim}
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\textbf{1 1 1 1 1 0 1 1 0 0 1 1 | Rn | (!= 1111 | Rd | 0 0 0 | M | Rm)} Ra
\end{verbatim}

\textbf{SMLAWB (M == 0)}

\begin{verbatim}
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
\end{verbatim}

\textbf{SMLAWT (M == 1)}

\begin{verbatim}
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
\end{verbatim}

if Ra == '1111' then SEE "SMULWB, SMULWT";
\begin{verbatim}
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
\end{verbatim}

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

\textbf{Assembler Symbols}

\begin{verbatim}
<	extit{c}>  See Standard assembler syntax fields.
<	extit{q}>  See Standard assembler syntax fields.
<Rd>  Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>  Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>  Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <	extit{y}>), encoded in the "Rm" field.
\end{verbatim}
<Ra> is the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(R[n]) * SInt(operand2) + (SInt(R[a]) << 16);
    R[d] = result<47:16>;
    if (result >> 16) != SInt(R[d]) then  // Signed overflow
        PSTATE.Q = '1';
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets **PSTATE.Q** to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | Rd | != 1111 | Rm | 0 | 1 | M | 1 | Rn |
| cond | Ra |

**SMLSD (M == 0)**

SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

**SMLSDX (M == 1)**

SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

if Ra == '1111' then SEE "SMUSD";

d = UInt(Rd);

n = UInt(Rn);

m = UInt(Rm);

a = UInt(Ra);

m_swap = (M == '1');

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | Rn | != 1111 | Rd | 0 | 0 | 0 | M | Rm |
| Ra |

**SMLSD (M == 0)**

SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

**SMLSDX (M == 1)**

SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

if Ra == '1111' then SEE "SMUSD";

d = UInt(Rd);

n = UInt(Rn);

m = UInt(Rm);

a = UInt(Ra);

m_swap = (M == '1');

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained unpredictable behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

### Assembler Symbols

- **<c>** See [Standard assembler syntax fields](#).
- **<q>** See [Standard assembler syntax fields](#).
- **<Rd>** Is the general-purpose destination register, encoded in the "Rd" field.
- **<Rn>** Is the first general-purpose source register, encoded in the "Rn" field.
- **<Rm>** Is the second general-purpose source register, encoded in the "Rm" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m], 16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then // Signed overflow
        PSTATE.Q = '1';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Multiply Subtract Long Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo $2^{64}$.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| != 1111                         | 0 1 1 1 0 1 0 0 RdHi            | RdLo                           | Rm                             | 0 1 M 1 Rn                      |

cond

SMLSLD (M == 0)

SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLSLDX (M == 1)

SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 1 1 0 1 1 1 1 0 1 Rn      | RdLo                           | RdHi                           | 1 1 0 M Rm                      |

SMLSLD (M == 0)

SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLSLDX (M == 1)

SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as NOP.
• The value in the destination register is UNKNOWN.

For more information about the CONstrained UNPREDICTable behavior of this instruction, see Architectural Constraints on UNPREDICTable behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<RdLo> Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SMMLA, SMMLAR**

Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value. Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd != 1111</td>
<td>SMMLA(R == 0)</td>
</tr>
<tr>
<td>Rm != 1111</td>
<td>SMMLA(R == 1)</td>
</tr>
<tr>
<td>Ra</td>
<td>SMMLA(R == 1)</td>
</tr>
</tbody>
</table>

### T1

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd != 1111</td>
<td>SMMLA(R == 0)</td>
</tr>
<tr>
<td>Rm != 1111</td>
<td>SMMLA(R == 1)</td>
</tr>
<tr>
<td>Ra</td>
<td>SMMLA(R == 1)</td>
</tr>
</tbody>
</table>

For more information about the constrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>`: See *Standard assembler syntax fields*.
- `<q>`: See *Standard assembler syntax fields*.
- `<Rd>`: Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>`: Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Rm>`: Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Ra>`: Is the third general-purpose source register holding the addend, encoded in the "Ra" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SMMLS, SMMLSR**

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the result of the subtraction before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------|-----------------|-----------------|-----------------|-----------------|
| != 1111                                  | 0   1           | 1               | 0               | 1               |
| cond                                    | Rd  | Ra  | Rm  | 1   1   | R  | 1   | Rn |

**SMMLS (R == 0)**

SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

**SMMLSR (R == 1)**

SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

\[
\begin{align*}
d &= \text{UInt}(Rd); \\
n &= \text{UInt}(Rn); \\
m &= \text{UInt}(Rm); \\
a &= \text{UInt}(Ra); \\
\text{round} &= (R == '1'); \\
\text{if } d == 15 || n == 15 || m == 15 || a == 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------|-----------------|-----------------|-----------------|
| 1   1           | 1   1           | 0   1           | 1   1           |
| Rn  | Ra  | Rd  | 0   0   | R  | Rm |

**SMMLS (R == 0)**

SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

**SMMLSR (R == 1)**

SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

\[
\begin{align*}
d &= \text{UInt}(Rd); \\
n &= \text{UInt}(Rn); \\
m &= \text{UInt}(Rm); \\
a &= \text{UInt}(Ra); \\
\text{round} &= (R == '1'); \\
\text{if } d == 15 || n == 15 || m == 15 || a == 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]

// Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Ra>` Is the third general-purpose source register holding the addend, encoded in the "Ra" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>cond</th>
<th>Rd</th>
<th>Rm</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>0 1 1 1 0 1 0 1</td>
<td>1 1 1</td>
<td>0 0</td>
</tr>
</tbody>
</table>

SMMUL (R == 0)

SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

SMMULR (R == 1)

SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

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

SMMUL (R == 0)

SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

SMMULR (R == 1)

SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.
Operation

if \texttt{ConditionPassed()} then
    \texttt{EncodingSpecificOperations();}
    \texttt{result = SInt(R[n]) * SInt(R[m]);}
    \texttt{if round then result = result + 0x80000000;}
    \texttt{R[d] = result<63:32>};

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMUAD, SMUADX

Signed Dual Multiply Add performs two signed 16 x 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the addition overflows. The multiplications cannot overflow.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

For more information about the CONSTRUED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  operand2 = if m_swap then ROR(R[m],16) else R[m];
  product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
  product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
  result = product1 + product2;
  R[d] = result<31:0>;
  if result != SInt(result<31:0>) then  // Signed overflow
    PSTATE.Q = '1';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| ! | 0 | 0 | 0 | 1 | 1 | 1 | Rd | (0) (0) (0) | Rm | 1 | M | N | 0 | Rn |
|---|---|---|---|---|---|---|----|---------|----|---|---|---|---|
| cond|

**SMULBB (M == 0 & N == 0)**

```
SMULBB{<c>}{<q}> {<Rd>,} <Rn>, <Rm>
```

**SMULBT (M == 1 & N == 0)**

```
SMULBT{<c>}{<q}> {<Rd>,} <Rn>, <Rm>
```

**SMULTB (M == 0 & N == 1)**

```
SMULTB{<c>}{<q}> {<Rd>,} <Rn>, <Rm>
```

**SMULTT (M == 1 & N == 1)**

```
SMULTT{<c>}{<q}> {<Rd>,} <Rn>, <Rm>
```

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[
n\text{ high} = (N == '1'); \quad m\text{ high} = (M == '1');
\]
\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0</td>
<td>1 1 1</td>
<td>Rd</td>
<td>0 0</td>
<td>N</td>
<td>M</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SMULBB (N == 0 & M == 0)**

```
SMULBB{<c>}{<q}> {<Rd>,} <Rn>, <Rm>
```

**SMULBT (N == 0 & M == 1)**

```
SMULBT{<c>}{<q}> {<Rd>,} <Rn>, <Rm>
```

**SMULTB (N == 1 & M == 0)**

```
SMULTB{<c>}{<q}> {<Rd>,} <Rn>, <Rm>
```

**SMULTT (N == 1 & M == 1)**

```
SMULTT{<c>}{<q}> {<Rd>,} <Rn>, <Rm>
```

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[
n\text{ high} = (N == '1'); \quad m\text{ high} = (M == '1');
\]
\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by `<x>`), encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the "Rm" field.

### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2);
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

### Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result. In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111| 0  | 0  | 0  | 0  | 1  | 1  | 0  | S  | RdHi| RdLo| Rm | 1  | 0  | 0  | 1  | Rn |

Flag setting (S == 1)

SMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Not flag setting (S == 0)

SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

\[
\begin{align*}
\text{dLo} &= \mathbf{UInt}(\text{RdLo}); \\
\text{dHi} &= \mathbf{UInt}(\text{RdHi}); \\
\text{n} &= \mathbf{UInt}(\text{Rn}); \\
\text{m} &= \mathbf{UInt}(\text{Rm}); \\
\text{setflags} &= (S == '1'); \\
\text{if dLo == 15} &\ |\ \text{dHi == 15} &\ |\ \text{n == 15} &\ |\ \text{m == 15} \text{ then UNPREDICTABLE;} \\
\text{if dHi == dLo} &\ \text{then UNPREDICTABLE;} \\
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \text{dHi == dLo}, then one of the following behaviors must occur:

- The instruction is \text{UNDEFINED}.
- The instruction executes as \text{NOP}.
- The value in the destination register is \text{UNKNOWN}.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | Rn | RdLo| RdHi| 0  | 0  | 0  | 0  | Rm |

T1

SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

\[
\begin{align*}
\text{dLo} &= \mathbf{UInt}(\text{RdLo}); \\
\text{dHi} &= \mathbf{UInt}(\text{RdHi}); \\
\text{n} &= \mathbf{UInt}(\text{Rn}); \\
\text{m} &= \mathbf{UInt}(\text{Rm}); \\
\text{setflags} &= \text{FALSE}; \\
\text{if dLo == 15} &\ |\ \text{dHi == 15} &\ |\ \text{n == 15} &\ |\ \text{m == 15} \text{ then UNPREDICTABLE;} \\
// \text{Armv8-A removes UNPREDICTABLE for R13} \\
\text{if dHi == dLo} &\ \text{then UNPREDICTABLE;} \\
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \text{dHi == dLo}, then one of the following behaviors must occur:

- The instruction is \text{UNDEFINED}.
- The instruction executes as \text{NOP}.
- The value in the destination register is \text{UNKNOWN}.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c> See Standard assembler syntax fields.

&q> See Standard assembler syntax fields.

<RdLo> Is the general-purpose destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi> Is the general-purpose destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational Information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rd</td>
<td>(0)(0)(0)(0)</td>
<td>Rm</td>
<td>1</td>
<td>M</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cond

SMULWB (M == 0)

SMULWB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMULWT (M == 1)

SMULWT{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

\[\begin{align*}
\text{d} &= \text{UInt}(\text{Rd}); \\
\text{n} &= \text{UInt}(\text{Rn}); \\
\text{m} &= \text{UInt}(\text{Rm}); \\
\text{m}\text{ high} &= (M == '1'); \\
\text{if d} &= 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\end{align*}\]

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>M</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SMULWB (M == 0)

SMULWB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMULWT (M == 1)

SMULWT{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

\[\begin{align*}
\text{d} &= \text{UInt}(\text{Rd}); \\
\text{n} &= \text{UInt}(\text{Rn}); \\
\text{m} &= \text{UInt}(\text{Rm}); \\
\text{m}\text{ high} &= (M == '1'); \\
\text{if d} &= 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\end{align*}\]

For more information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by `<y>`), encoded in the "Rm" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m high then $R[m]<31:16>$ else $R[m]<15:0>$;
    product = $SInt(R[n]) * SInt(operand2);$;
    $R[d] = product<47:16>$;
    // Signed overflow cannot occur

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow cannot occur.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| ! = 1111 | 0 1 1 1 0 | 0 0 0 | Rd | 1 1 1 1 | Rm | 0 1 | M | 1 | Rn |
| cond |

SMUSD (M == 0)

SMUSD{<c>{<q>} {<Rd>},} <Rn>, <Rm>

SMUSDX (M == 1)

SMUSDX{<c>{<q>} {<Rd>},} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

| 1 1 1 1 1 0 1 1 0 | 1 0 0 | Rn | 1 1 1 1 | Rd | 0 0 0 | M | Rm |

SMUSD (M == 0)

SMUSD{<c>{<q>} {<Rd>},} <Rn>, <Rm>

SMUSDX (M == 1)

SMUSDX{<c>{<q>} {<Rd>},} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.

<cond> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.
if ConditionPassed() then
    EncodingSpecificOperations();
operand2 = if m_swap then ROR(R[m],16) else R[m];
product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
result = product1 - product2;
R[d] = result<31:0>;
// Signed overflow cannot occur
SRS, SRSDA, SRSDB, SRSIA, SRSIB

Store Return State stores the LR_<current_mode> and SPSR_<current_mode> to the stack of a specified mode. For information about memory accesses see Memory accesses.
SRS is UNDEFINED in Hyp mode.
SRS is CONSTRAINED UNPREDICTABLE if it is executed in User or System mode, or if the specified mode is any of the following:
- Not implemented.
- A mode that Table G1-5 does not show.
- Hyp mode.
- Monitor mode, if the SRS instruction is executed in Non-secure state.
If EL3 is using AArch64 and an SRS instruction that is executed in a Secure EL1 mode specifies Monitor mode, it is trapped to EL3.
See Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

```
31 30 29 28 27 26 25 24 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 1 1 1 0 0 | P | U | W | 0 | (1) | 0 | (1) | 0 | (1) | 0 | (0) | 0 | (0) | 0 | (0) | 0 | (0) | 0 | (0) | 0 | (0) | 0 | mode
```

Decrement After (P == 0 && U == 0)

SRSDA{<c>}{<q>} SP{!}, #<mode>

Decrement Before (P == 1 && U == 0)

SRSDB{<c>}{<q>} SP{!}, #<mode>

Increment After (P == 0 && U == 1)

SRS{IA}{<c>}{<q>} SP{!}, #<mode>

Increment Before (P == 1 && U == 1)

SRSIB{<c>}{<q>} SP{!}, #<mode>

wback = (W == '1'); increment = (U == '1'); wordhigher = (P == U);

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 0 0 0 | 0 | 0 | 0 | 0 | W | 0 | (1) | 0 | (1) | 0 | (1) | 0 | (1) | 0 | (1) | 0 | (0) | 0 | (1) | 0 | (1) | 0 | (0) | 0 | mode
```

T1

SRSDB{<c>}{<q>} SP{!}, #<mode>

wback = (W == '1'); increment = FALSE; wordhigher = FALSE;

T2

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 0 0 | 1 | 1 | 0 | W | 0 | (1) | 0 | (1) | 0 | (1) | 0 | (1) | 0 | (1) | 0 | (0) | 0 | (0) | 0 | (1) | 0 | mode
```

SRS, SRSDA, SRSDB, SRSIA, SRSIB
SRS{IA}{<c>}{<q>}{SP{!}}, #<mode>

\[
\text{wback} = (W == '1'); \quad \text{increment} = \text{TRUE}; \quad \text{wordhigher} = \text{FALSE};
\]

For more information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly SRS (T32) and SRS (A32).

**Assembler Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA</td>
<td>For encoding A1: is an optional suffix to indicate the Increment After variant. For encoding T2: is an optional suffix for the Increment After form.</td>
</tr>
<tr>
<td>&lt;c&gt;</td>
<td>For encoding A1: see Standard assembler syntax fields. &lt;c&gt; must be AL or omitted. For encoding T1 and T2: see Standard assembler syntax fields.</td>
</tr>
<tr>
<td>&lt;q&gt;</td>
<td>See Standard assembler syntax fields.</td>
</tr>
<tr>
<td>!</td>
<td>The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the &quot;W&quot; field as 1, otherwise this field defaults to 0.</td>
</tr>
<tr>
<td>&lt;mode&gt;</td>
<td>Is the number of the mode whose Banked SP is used as the base register, encoded in the &quot;mode&quot; field. For details of PE modes and their numbers see AArch32 PE mode descriptions.</td>
</tr>
</tbody>
</table>

SRSFA, SRSEA, SRSFD, and SRSED are pseudo-instructions for SRSIB, SRSIA, SRSDB, and SRSDA respectively, referring to their use for pushing data onto Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.
if \textbf{CurrentInstrSet}() \textbf{==} \textbf{InstrSet.A32} then \\
if \textbf{ConditionPassed}() then \\
  \textbf{EncodingSpecificOperations}(); \\
  \textbf{if \ PSTATE.EL} \textbf{== EL2} then \textbf{// UNDEFINED at EL2 UNDEFINED};

// Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these 
// to be security holes
if \textbf{PSTATE.M \textbf{IN \{M32.User,M32.System\}} then 
  \textbf{UNPREDICTABLE};
elsif \textbf{mode} \textbf{== M32.Hyp} then \textbf{// Check for attempt to access Hyp mode SP}
  \textbf{UNPREDICTABLE};
elsif \textbf{mode} \textbf{== M32.Monitor} then \textbf{// Check for attempt to access Monitor mode SP}
  if \textbf{!HaveEL(EL3) || !IsSecure()} then 
    \textbf{UNPREDICTABLE};
  elsif \textbf{!ELUsingAArch32(EL3)} then 
    \textbf{AArch64.MonitorModeTrap}();
elsif \textbf{BadMode}(\textbf{mode}) then 
  \textbf{UNPREDICTABLE};

\textbf{base} = \textbf{Rmode}[13,mode];
\textbf{address} = \textbf{if increment then base else base-8};
\textbf{if wordhigher then address = address+4};
\textbf{MemA}[\textbf{address},4] = LR;
\textbf{MemA}[\textbf{address+4},4] = SPSR[];
\textbf{if wback then Rmode}[13,mode] = \textbf{if increment then base+8 else base-8};

\textbf{else}
if \textbf{ConditionPassed}() then \\
  \textbf{EncodingSpecificOperations}();
if \textbf{PSTATE.EL} \textbf{== EL2} then \textbf{// UNDEFINED at EL2 UNDEFINED};

// Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these 
// to be security holes
if \textbf{PSTATE.M \textbf{IN \{M32.User,M32.System\}} then 
  \textbf{UNPREDICTABLE};
elsif \textbf{mode} \textbf{== M32.Hyp} then \textbf{// Check for attempt to access Hyp mode SP}
  \textbf{UNPREDICTABLE};
elsif \textbf{mode} \textbf{== M32.Monitor} then \textbf{// Check for attempt to access Monitor mode SP}
  if \textbf{!HaveEL(EL3) || !IsSecure()} then 
    \textbf{UNPREDICTABLE};
  elsif \textbf{!ELUsingAArch32(EL3)} then 
    \textbf{AArch64.MonitorModeTrap}();
elsif \textbf{BadMode}(\textbf{mode}) then 
  \textbf{UNPREDICTABLE};

\textbf{base} = \textbf{Rmode}[13,mode];
\textbf{address} = \textbf{if increment then base else base-8};
\textbf{if wordhigher then address = address+4};
\textbf{MemA}[\textbf{address},4] = LR;
\textbf{MemA}[\textbf{address+4},4] = SPSR[];
\textbf{if wback then Rmode}[13,mode] = \textbf{if increment then base+8 else base-8};

\textbf{CONSTRAINED UNPREDICTABLE behavior}

\textbf{If PSTATE.M \textbf{IN \{M32.User,M32.System\}}}, then one of the following behaviors must occur:

- \textbf{The instruction is UNDEFINED}.
- \textbf{The instruction executes as NOP}.

\textbf{If mode == M32.Hyp}, then one of the following behaviors must occur:

- \textbf{The instruction is UNDEFINED}.
- \textbf{The instruction executes as NOP}.

\textbf{If mode == M32.Monitor \&\& (!HaveEL(EL3) || !IsSecure())}, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.

If BadMode(mode), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction stores to the stack of the mode in which it is executed.
• The instruction stores to an UNKNOWN address, and if the instruction specifies writeback then any general-purpose register that can be accessed from the current Exception level without a privilege violation becomes UNKNOWN.
SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.
This instruction sets PSTATE.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--------------|--------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| != 1111 | 0 | 1 | 0 | 1 | 0 | 1 | sat_imm | Rd | imm5 | sh | 0 | 1 | Rn |

cond

Arithmetic shift right (sh == 1)

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Logical shift left (sh == 0)

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--------------|--------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 | 1 | 1 | 1 | 0 | [0] | 1 | 1 | 0 | 0 | sh | 0 | Rn | 0 | imm3 | Rd | imm2|[0] | sat_imm |

Arithmetic shift right (sh == 1 && !(imm3 == 000 && imm2 == 00))

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Logical shift left (sh == 0)

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

if sh == '1' && (imm3:imm2) == '0000' then SEE "SSAT16";

d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5:imm2);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<imm> Is the bit position for saturation, in the range 1 to 32, encoded in the "sat_imm" field as <imm>-1.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

<amount> For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32.
For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.

For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    (result, sat) = SignedSatQ(SInt(operand), saturate_to);
    R[d] = SignExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range. This instruction sets PSTATE.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

!= 1111 | 0 1 1 0 1 0 1 0 | sat_imm | Rd | (1)(1)(1)(1) | 0 0 1 1 | Rn
```

```
cond
```

```
A1

SSAT16{<c}>{<q>} <Rd>, #<imm>, <Rn>

d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1 1 1 1 0 |0| 1 1 0 0 1 0 | Rn | 0 0 0 0 | Rd | 0 0 |0|0|0|0| sat_imm
```

```
T1

SSAT16{<c}>{<q>} <Rd>, #<imm>, <Rn>

d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<imm>` Is the bit position for saturation, in the range 1 to 16, encoded in the "sat_imm" field as `<imm>`-1.
- `<Rn>` Is the general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then

  EncodingSpecificOperations();
  (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
  (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
  R[d]<15:0> = SignExtend(result1, 16);
  R[d]<31:16> = SignExtend(result2, 16);
  if sat1 || sat2 then
    PSTATE.Q = '1';
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets PSTATE.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| ![SSAX](image) |

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| != 1111 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | Rn | Rd | (1)(1)(1)(1) | 0 | 1 | 0 | 1 | Rm |

cond

A1

SSAX{<c>}{<q>} {<Rd>,} {<Rn>,} <Rm>

```plaintext
 d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

| ![T1](image) |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 0 | 0 | 0 | Rm |

T1

SSAX{<c>}{<q>} {<Rd>,} {<Rn>,} <Rm>

```plaintext
 d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    sum = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0> = sum<15:0>;
    R[d]<31:16> = diff<15:0>;
    PSTATE.GE<1:0> = if sum >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if diff >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination.
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

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

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

SSBB{<q>}

// No additional decoding required

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------------------|
| 1 1 1 0 0 1 1 1 0 1 1 (1)(1)(1)(1) 1 0 (0) 0 (1)(1)(1)(1) 0 1 0 0 0 0 0 0 |

T1

SSBB{<q>}

if InITBlock() then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q> See Standard assembler syntax fields.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  SpeculativeStoreBypassBarrierToVA();
SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | Rn | Rd | (1)(1)(1)(1) | 0 | 1 | 1 | 1 | Rm |

cond

A1

SSUB16{<c>}{<q>} {<Rd>},{<Rn>,}<Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

T1

SSUB16{<c>}{<q>} {<Rd>},{<Rn>,}<Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
  diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
  R[d]<15:0> = diff1<15:0>;
  R[d]<31:16> = diff2<15:0>;
  PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
  PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SSUB8

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| != 1111 | 0 1 1 0 0 0 1 | Rn | Rd | (1) | (1) | (1) | 1 1 1 1 | Rm |
|cond|

A1

SSUB8{<c>}{<q>}{<Rd>},{<Rn>},{<Rm>}

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

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

T1

SSUB8{<c>}{<q>}{<Rd>},{<Rn>},{<Rm>}

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0> = diff1<7:0>;
    R[d]<15:8> = diff2<7:0>;
    R[d]<23:16> = diff3<7:0>;
    R[d]<31:24> = diff4<7:0>;
    PSTATE.GE<0> = if diff1 >= 0 then '1' else '0';
    PSTATE.GE<1> = if diff2 >= 0 then '1' else '0';
    PSTATE.GE<2> = if diff3 >= 0 then '1' else '0';
    PSTATE.GE<3> = if diff4 >= 0 then '1' else '0';
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Store data to System register calculates an address from a base register value and an immediate offset, and stores a word from the DBGDTRRXint System register to memory. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses see Memory accesses.

In an implementation that includes EL2, the permitted STC access to DBGDTRRXint can be trapped to Hyp mode, meaning that an attempt to execute an STC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see Trapping general Non-secure System register accesses to debug registers.

For simplicity, the STC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

Offset (P == 1 && W == 0)

STC{<c>{<q>}} p14, c5, [<Rn>{, #<+/-><imm>}]}

Post-indexed (P == 0 && W == 1)

STC{<c>{<q>}} p14, c5, [<Rn>], #<+/-><imm>

Pre-indexed (P == 1 && W == 1)

STC{<c>{<q>}} p14, c5, [<Rn>, #<+/-><imm>]

Unindexed (P == 0 && U == 1 && W == 0)

STC{<c>{<q>}} p14, c5, [<Rn>], <option>

if P == '0' && U == '0' && W == '0' then UNDEFINED;

n = UINT(Rn);  cp = 14;

imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');

if n == 15 && wback || CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If n == 15 && wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 0</td>
</tr>
</tbody>
</table>
Offset (P == 1 && W == 0)

\[ \text{STC}\{c\}\{q\} \text{ p14, c5, } [\text{<Rn>}, \{+/\-\}<\text{imm}>] \]

Post-indexed (P == 0 && W == 1)

\[ \text{STC}\{c\}\{q\} \text{ p14, c5, } [\text{<Rn}>], \{+/\-\}<\text{imm}> \]

Pre-indexed (P == 1 && W == 1)

\[ \text{STC}\{c\}\{q\} \text{ p14, c5, } [\text{<Rn>}, \{+/\-\}<\text{imm}>]! \]

Unindexed (P == 0 && U == 1 && W == 0)

\[ \text{STC}\{c\}\{q\} \text{ p14, c5, } [\text{<Rn>}, <\text{option}> \text{ if } P == '0' \&\& U == '0' \&\& W == '0' \text{ then UNDEFINED}; \]
\[ n = \text{UInt}(Rn); \; cp = 14; \]
\[ \text{imm32} = \text{ZeroExtend}(\text{imm8:'00', 32}); \; \text{index} = (P == '1'); \; \text{add} = (U == '1'); \; \text{wback} = (W == '1'); \]
\[ \text{if } n == 15 \&\& (\text{wback || CurrentInstrSet() != InstrSet_A32}) \text{ then UNPREDICTABLE;} \]

CONSTRANDED UNPREDICTABLE behavior

If n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rn>` For the offset or unindexed variant: is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
  
  For the offset, post-indexed or pre-indexed variant: is the general-purpose base register, encoded in the "Rn" field.
- `<option>` Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the “imm8” field. The value of this field is ignored when executing this instruction.
- `+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th></th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

- `<imm>` Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the “imm8” field, as `<imm>/4`.

Operation

if \text{ConditionPassed()} \text{ then}
\[ \text{EncodingSpecificOperations();} \]
\[ \text{offset_addr} = \text{if add then } (R[n] + \text{imm32}) \text{ else } (R[n] - \text{imm32}); \]
\[ \text{address} = \text{if index then offset_addr else } R[n]; \]
\[ // \text{System register read from DBGDTRRXint.} \]
\[ \text{MemA}[\text{address},4] = \text{DBGDTR_EL0}[]; \]
\[ \text{if wback then } R[n] = \text{offset_addr}; \]
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STL

Store-Release Word stores a word from a register to memory. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.
For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-----------------|----------------|-----------------|-----------------|-----------------|-----------------|
| != 1111           | 0 0 0 1 1 0 0 0  | Rn              | (1)(1)(1)(1)(1)| (1)            | 0 0 1 0 0 1    | Rt              |
| cond              |

\[
\text{STL}\{<c>\}{<q>}\{<Rt>\}, [<Rn>] \\
t = \text{UInt}(Rt); n = \text{UInt}(Rn);
\text{if } t == 15 \text{ || } n == 15 \text{ then UNPREDICTABLE;}
\]

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 0 0 1 1 0 0</td>
</tr>
</tbody>
</table>

\[
\text{STL}\{<c>\}{<q>}\{<Rt>\}, [<Rn>] \\
t = \text{UInt}(Rt); n = \text{UInt}(Rn);
\text{if } t == 15 \text{ || } n == 15 \text{ then UNPREDICTABLE;}
\]

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation

if \text{ConditionPassed}() \text{ then}
   \text{EncodingSpecificOperations();}
   \text{address} = R[n];
   \text{Mem0}[\text{address}, 4] = R[t];

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLB

Store-Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
! = 1111 0 0 0 1 1 1 0 0   Rn  (1)(1)(1)(1)(1) 0 0 1 0 0 1   Rt
```

cond

```
A1

STLB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 0 0 0 1 1 0 0   Rn   Rt  (1)(1)(1)(1) 1 0 0 0 (1)(1)(1)(1)
```

T1

```
T1

STLB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemQ[address, 1] = R[t]<7:0>;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLEX

Store-Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>! 1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td>(1)</td>
<td>(1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**cond**

STLEX{<c>}{<q>} {<Rd>, <Rt>, [<Rn>]}  

d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(d == t\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \(d == n\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The store instruction executes but the value stored is **UNKNOWN**.

### T1

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rt</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

STLEX{<c>}{<q>} {<Rd>, <Rt>, [<Rn>]}  

d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(d == t\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \(d == n\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
• The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<\texttt{c}> See Standard assembler syntax fields.

<\texttt{q}> See Standard assembler syntax fields.

<\texttt{Rd}> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<\texttt{Rt}> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<\texttt{Rn}> Is the general-purpose base register, 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.
• <\texttt{Rd}> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

• If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
• Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.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

\begin{verbatim}
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemO[address, 4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
\end{verbatim}

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLEXB

Store-Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ! = 1111 | 0 | 0 | 1 | 1 | 0 | 0 | Rn | Rd | (1) | (1) | 0 | 1 | 0 | 0 | 1 | Rt |

cond

A1

`STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]`

\[ d = \text{UInt}(Rd); \ t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \]
\[ \text{if} \ d == 15 || \ t == 15 || \ n == 15 \text{ then UNPREDICTABLE}; \]
\[ \text{if} \ d == n || d == t \text{ then UNPREDICTABLE}; \]

CONSTRUCTED UNPREDICTABLE behavior

If \( d \equiv t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( d \equiv n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Rn | Rt | (1) | (1) | (1) | 1 | 1 | 0 | 0 | Rd |

T1

`STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]`

\[ d = \text{UInt}(Rd); \ t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \]
\[ \text{if} \ d == 15 || \ t == 15 || \ n == 15 \text{ then UNPREDICTABLE}; \]
\[ \text{if} \ d == n || d == t \text{ then UNPREDICTABLE}; \]

CONSTRUCTED UNPREDICTABLE behavior

If \( d \equiv t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( d \equiv n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0  If the operation updates memory.

1  If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If AArch32.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

\[
\text{if } \text{ConditionPassed}() \text{ then}
\text{EncodingSpecificOperations();}
\text{address} = \text{R}[n];
\text{if } \text{AArch32.ExclusiveMonitorsPass(address,} 1) \text{ then}
\text{Mem0}[address, 1] = \text{R}[t]<7:0>;
\text{R}[d] = \text{ZeroExtend}'0';
\text{else}
\text{R}[d] = \text{ZeroExtend}'1';
\]

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLEXD

Store-Release Exclusive Doubleword stores a doubleword from two registers to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
! = 1111 | 0 0 0 1 1 0 1 0 | Rn | Rd | (1)(1) | 1 0 | 1 0 0 1 | Rt
cond
```

A1

```
STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

\[
d = \text{UInt}(Rd); \quad t = \text{UInt}(Rt); \quad t2 = t+1; \quad n = \text{UInt}(Rn);
\]
\[
\text{if } d == 15 \mid \mid Rt<0> == '1' \mid \mid t2 == 15 \mid \mid n == 15 \text{ then UNPREDICTABLE;}
\]
\[
\text{if } d == n \mid \mid d == t \mid \mid d == t2 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(d == t\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \(d == n\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs the store to an **UNKNOWN** address.

If \(Rt<0> == '1'\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes with the additional decode: \(Rt<0> == '0'\).
- The instruction executes with the additional decode: \(t2 = t\).
- The instruction executes as described, with no change to its behavior and no additional side effects.

If \(Rt == '1110'\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction is handled as described in **Using R15**.

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 0 0 0 1 1 0 0 Rn | Rt | Rt2 | 1 1 1 1 | Rd
```
STLEXD\(<c>\){<q>}, \(<Rd>\), \(<Rt>\), \(<Rt2>\), \([<Rn>]\)

\[
d = \text{UInt}(Rd); \quad t = \text{UInt}(Rt); \quad t2 = \text{UInt}(Rt2); \quad n = \text{UInt}(Rn);
\]

if \(d == 15 || t == 15 || t2 == 15 || n == 15\) then UNPREDICTABLE;
if \(d == n || d == t || d == t2\) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(d == t\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \(d == n\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

\(<c>\) See *Standard assembler syntax fields*.

\(<q>\) See *Standard assembler syntax fields*.

\(<Rd>\) Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

\(<Rt>\) For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rt2>\) For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

\(<Rn>\) Is the general-purpose base register, 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.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.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

if ConditionPassed() then
    EncodingSpecificOperations();
    address = B[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian(AccType_ORDERED) then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address, 8) then
        MemO[address, 8] = value;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**STLEXH**

Store-Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ! = 1111 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rn | Rd | (1) | (1) | 1 | 0 | 1 | 0 | 0 | 1 | Rt |

**cond**

#### A1

\[
\text{STLEXH}\{<c>\}\{<q>\} <Rd>, <Rt>, [<Rn>]
\]

\[
d = \text{UInt}(Rd); \quad t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

\[
\text{if } d == 15 \text{ or } t == 15 \text{ or } n == 15 \text{ then UNPREDICTABLE;}
\]

\[
\text{if } d == n \text{ or } d == t \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d == t \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \( d == n \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs the store to an **UNKNOWN** address.

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rt</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

#### T1

\[
\text{STLEXH}\{<c>\}\{<q>\} <Rd>, <Rt>, [<Rn>]
\]

\[
d = \text{UInt}(Rd); \quad t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

\[
\text{if } d == 15 \text{ or } t == 15 \text{ or } n == 15 \text{ then UNPREDICTABLE;}
\]

\[
\text{if } d == n \text{ or } d == t \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d == t \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \( d == n \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
• The instruction executes as NOP.
• The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

 0   If the operation updates memory.

 1   If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, 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
  • <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
  • If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
  • Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.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

if ConditionPassed() then
  EncodingSpecificOperations();
  address = R[n];
  if AArch32.ExclusiveMonitorsPass(address,2) then
    MemO[address, 2] = R[t]<15:0>;
    R[d] = ZeroExtend('0');
  else
    R[d] = ZeroExtend('1');

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLH

Store-Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
! = 1111 0 0 0 1 1 1 0 Rn (1)(1)(1)(1)(1)(1) 0 0 1 0 0 1 Rt

cond
```

A1

```
STLH{<c>}{<q>}{<Rt>, [<Rn]>}
```

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 0 0 0 1 1 0 0 Rn (1)(1)(1)(1)(1) 0 0 1 0 0 1 (1)(1)(1)(1)(1)
```

T1

```
STLH{<c>}{<q>}{<Rt>, [<Rn]>}
```

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 2] = R[t]<15:0>;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STM, STMIA, STMEA

Store Multiple (Increment After, Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC. Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

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

cond

!= 1111 | 1 0 0 0 | 1 0 | W | 0 | Rn | register_list
```

\[
\text{A1}
\]

\[
\text{STM\{IA\}{<c}\}{<q}\}{<Rn>!}, \ <\text{registers}> \ // \ (\text{Preferred \ syntax})
\]

\[
\text{STMEA\{<c}\}{<q}\}{<Rn>!}, \ <\text{registers}> \ // \ (\text{Alternate \ syntax, \ Empty \ Ascending \ stack})
\]

\[
n = \text{UInt}(Rn); \ \text{registers} = \text{register_list}; \ \text{wback} = (W == '1');
\]

\[
\text{if} \ n == 15 || \ \text{BitCount}(\text{registers}) < 1 \ \text{then} \ \text{UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If n == 15 & wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

### T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1 1 0 0 0 0 | Rn | register_list
```

\[
\text{T1}
\]

\[
\text{STM\{IA\}{<c}\}{<q}\}{<Rn>!}, \ <\text{registers}> \ // \ (\text{Preferred \ syntax})
\]

\[
\text{STMEA\{<c}\}{<q}\}{<Rn>!}, \ <\text{registers}> \ // \ (\text{Alternate \ syntax, \ Empty \ Ascending \ stack})
\]

\[
n = \text{UInt}(Rn); \ \text{registers} = '00000000':\text{register_list}; \ \text{wback} = \text{TRUE};
\]

\[
\text{if} \ \text{BitCount}(\text{registers}) < 1 \ \text{then} \ \text{UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
The instruction executes as **NOP**.
The instruction operates as an **STM** with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If \( n == 15 && \text{wback} \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15*.

### T2

![Register Configuration](image)

**STM**(IA){<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be represented in T1)

**STMEA**(IA){<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack, if <Rn>, '!' and <registers> can be represented in T1)

**STM**(IA){<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

**STMEA**(IA){<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)

\[
n = \text{UInt}(Rn); \quad \text{registers} = P:M:\text{register\_list}; \quad \text{wback} = (W == '1');\]

If \( n == 15 || \text{BitCount}(\text{registers}) < 2 \), then UNPREDICTABLE;

if \( \text{wback} && \text{registers}<n> == '1' \), then UNPREDICTABLE;

if \( \text{registers}<13> == '1' \), then UNPREDICTABLE;

if \( \text{registers}<15> == '1' \), then UNPREDICTABLE;

---

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{BitCount}(\text{registers}) < 1 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction operates as an **STM** with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If \( \text{BitCount}(\text{registers}) == 1 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an **STM** with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If \( \text{wback} && \text{registers}<n> == '1' \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored for the base register is **UNKNOWN**.

If \( \text{registers}<13> == '1' \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is **UNKNOWN**.

If \( \text{registers}<15> == '1' \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
• The instruction executes as NOP.
• The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

If \( n \equiv 15 \land wback \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

IA Is an optional suffix for the Increment After form.
<c> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.
<Rn> Is the general-purpose base register, encoded in the “Rn” field.
! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.
<registers> For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by \{ and \}. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.
If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by \{ and \}. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by \{ and \}. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN; // Only possible for encodings T1 and A1
            else
                MemA[address,4] = R[i];
                address = address + 4;
        if registers<15> == '1' then // Only possible for encoding A1
            MemA[address,4] = PCStoreValue();
            if wback then R[n] = R[n] + 4*BitCount(registers);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STM (User registers)

In an EL1 mode other than System mode, Store Multiple (User registers) stores multiple User mode registers to consecutive memory locations using an address from a base register. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Store Multiple (User registers) is UNDEFINED in Hyp mode, and CONSTRAINED UNPREDICTABLE in User or System modes.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC.

A1

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

```
STM{<amode>}{<c>}{<q>} {<Rn>, <registers>}

n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

```
<amode> is one of:

DA  Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.

ED  Empty Descending. For this instruction, a synonym for DA.

DB  Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.

FD  Full Descending. For this instruction, a synonym for DB.

IA  Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.

EA  Empty Ascending. For this instruction, a synonym for IA.

IB  Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.
```
FA

Full Ascending. For this instruction, a synonym for IB.

See Standard assembler syntax fields.

See Standard assembler syntax fields.

<Rn>
Is the general-purpose base register, encoded in the "Rn" field.

<registers>
Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the STM instruction. The registers are stored with the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  if PSTATE.EL == EL2 then
    UNDEFINED;
  elsif PSTATE.M IN {M32_User, M32_System} then
    UNPREDICTABLE;
  else
    length = 4*BitCount(registers);
    address = if increment then R[n] else R[n]-length;
    if wordhigher then address = address+4;
    for i = 0 to 14
      if registers<i> == '1' then // Store User mode register
        MemA[address,4] = Rmode[i, M32_User];
        address = address + 4;
      if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();

CONstrained UNPREDICTABLE behavior

If PSTATE.M IN {M32_User, M32_System}, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STMDA, STMED

Store Multiple Decrement After (Empty Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC. Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

A1

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>1111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>W</td>
<td>0</td>
<td>Rn</td>
<td>register_list</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

A1

STMDA{<c>}{<q>}{<Rn>}{!}, <registers> // (Preferred syntax)
STMED{<c>}{<q>}{<Rn>}{!}, <registers> // (Alternate syntax, Empty Descending stack)

\[
n = \text{UInt}(Rn); \ \text{registers} = \text{register_list}; \ \text{wback} = (W == '1');
\]

if \( n == 15 || \text{BitCount}(\text{registers}) < 1 \) then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If \( \text{BitCount}(\text{registers}) < 1 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If \( n == 15 && \text{wback} \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
- `<registers>` Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.
if ConditionPassed() then
    EncodingSpecificOperations();
    address = \texttt{R}[n] - 4*\texttt{BitCount}(\texttt{registers}) + 4;
    for i = 0 to 14
        if registers\texttt{<i>} == '1' then
            if i == n && \texttt{wback} && i != LowestSetBit(\texttt{registers}) then
                \texttt{MemA}[\texttt{address},4] = \texttt{bits}(32) UNKNOWN;
            else
                \texttt{MemA}[\texttt{address},4] = \texttt{R}[i];
                address = address + 4;
        if registers\texttt{<15>} == '1' then
            \texttt{MemA}[\texttt{address},4] = \texttt{PCStoreValue}();
        if \texttt{wback} then \texttt{R}[n] = \texttt{R}[n] - 4*\texttt{BitCount}(\texttt{registers});

\textbf{Operational information}

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STMDB, STMFD

Store Multiple Decrement Before (Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

This instruction is used by the alias PUSH (multiple registers).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| != 1111 | 1 0 0 | 1 0 0 | W | 0 | Rn | register_list |
```

**T1**

```
<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 0</td>
</tr>
</tbody>
</table>
```

**CONSTRANDED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

**T1**

```
<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 0</td>
</tr>
</tbody>
</table>
```

**CONSTRANDED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as **NOP**.
• The instruction operates as an **STM** with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If $\text{wback} \&\& \text{registers}<n> == '1'$, then one of the following behaviors must occur:

• The instruction is **UNDEFINED**.
• The instruction executes as **NOP**.
• The store instruction executes but the value stored for the base register is **UNKNOWN**.

If $\text{BitCount(registers)} == 1$, then one of the following behaviors must occur:

• The instruction is **UNDEFINED**.
• The instruction executes as **NOP**.
• The instruction executes as described, with no change to its behavior and no additional side effects.
• The store instruction performs all of the stores using the specified addressing mode but the value of R15 is **UNKNOWN**.

If $\text{registers}<13> == '1'$, then one of the following behaviors must occur:

• The instruction is **UNDEFINED**.
• The instruction executes as **NOP**.
• The instruction executes as described, with no change to its behavior and no additional side effects.
• The store instruction performs all of the stores using the specified addressing mode but the value of R13 is **UNKNOWN**.

If $\text{registers}<15> == '1'$, then one of the following behaviors must occur:

• The instruction is **UNDEFINED**.
• The instruction executes as **NOP**.
• The store instruction performs all of the stores using the specified addressing mode but the value of R15 is **UNKNOWN**.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

<\text{c}>
See *Standard assembler syntax fields*.

<\text{q}>
See *Standard assembler syntax fields*.

<Rn>
Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<\text{registers}>
For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.
The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.
If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an **UNKNOWN** value for the base register.
For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.
The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUSH (multiple registers)</strong></td>
<td>T1</td>
<td>$W == '1' &amp;&amp; Rn == '1101' &amp;&amp; \text{BitCount(M:register_list)} &gt; 1</td>
</tr>
<tr>
<td><strong>PUSH (multiple registers)</strong></td>
<td>A1</td>
<td>$W == '1' &amp;&amp; Rn == '1101' &amp;&amp; \text{BitCount(register_list)} &gt; 1</td>
</tr>
</tbody>
</table>
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    address = \texttt{R[n]} - 4*\texttt{BitCount}(\texttt{registers});
    for i = 0 to 14
        if registers\textless{}i\textgreater{} == '1' then
            if i == n \&\& wback \&\& i != LowestSetBit(\texttt{registers}) then
                \texttt{MemA[address,4]} = \texttt{bits(32) UNKNOWN};  // Only possible for encoding A1
            else
                \texttt{MemA[address,4]} = \texttt{R[i]};
                address = address + 4;
        if registers\textless{}15\textgreater{} == '1' then  // Only possible for encoding A1
            \texttt{MemA[address,4]} = \texttt{PCStoreValue()};
        if wback then \texttt{R[n]} = \texttt{R[n]} - 4*\texttt{BitCount}(\texttt{registers});

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STMIB, STMFA

Store Multiple Increment Before (Full Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

A1

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
! = 1111 | 1  0  0 | 1  0 | W | 0 | Rn | register list
```

cond

A1

STMIB{<c}{{<q}} {<Rn}{{!}, <registers}> // (Preferred syntax)
STMFA{<c}{{<q}} {<Rn}{{!}, <registers}> // (Alternate syntax, Full Ascending stack)

\[
n = \text{UInt}(Rn); \text{ registers } = \text{ register list}; \text{ wback } = (W == '1');\\
\text{if } n == 15 || \text{BitCount(registers) < 1 then UNPREDICTABLE;}
\]

CONstrained UNPredICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If n == 15 && wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONstrained UNPredICTABLE behavior of this instruction, see Architectural Constraints on UNPredICTABLE behaviors.

Assembler Symbols

- `<c>`: See Standard assembler syntax fields.
- `<q>`: See Standard assembler syntax fields.
- `<Rn>`: Is the general-purpose base register, encoded in the “Rn” field.
- `<registers>`: Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.
  The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.
  If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.
Operation

if `ConditionPassed()` then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
                address = address + 4;
        end if
    end for
    if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();
    end if
    if wback then
        R[n] = R[n] + 4*BitCount(registers);
    end if
end if

Operational information

If CPSR.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) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

This instruction is used by the alias PUSH (single register).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3 and T4).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 1 0</td>
</tr>
<tr>
<td>----------------</td>
</tr>
</tbody>
</table>

cond

Offset (P == 1 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn> {, #}]{+/-}imm]

Post-indexed (P == 0 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn>], #){+/-}imm>

Pre-indexed (P == 1 && W == 1)

STR{<c>}{<q>} <Rt>, [<Rn>], #){+/-}imm}!

if P == '0' && W == '1' then SEE "STRT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
if wback && n == 15 || n == t then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0</td>
</tr>
</tbody>
</table>

T1

STR{<c>}{<q>} <Rt>, [<Rn> {, #}]{+/-}imm]

t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:00, 32);
index = TRUE; add = TRUE; wback = FALSE;
T2

STR{<c>}{<q>}{<Rt>}, [SP{, #}{<mm>}] {Rt, imm8}

\(t = \text{UInt}(Rt); \ n = 13; \ \text{imm32} = \text{ZeroExtend}(\text{imm8}:'00', 32)\);
index = TRUE; add = TRUE; wback = FALSE;

T3

STR{<c>}{<q>}{<Rt>}, [SP{, #}{<mm>}] {Rt, imm12}

if \(Rn == '1111'\) then UNDEFINED;
\(t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \ \text{imm32} = \text{ZeroExtend}(\text{imm12}, 32)\);
index = TRUE; add = TRUE; wback = FALSE;
if \(t == 15\) then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \(t == 15\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T4

Offset (\(P == 1 \&\& U == 0 \&\& W == 0\))

\(\text{STR}(\text{immediate})\)
**CONSTRAINED UNPREDICTABLE behavior**

If \( w_{back} \) & \( n == t \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \( w_{back} \) & \( n == 15 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If \( t == 15 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction performs the store using the specified addressing mode but the value corresponding to \( R15 \) is **UNKNOWN**.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

\(<c>\) See **Standard assembler syntax fields**.

\(<q>\) See **Standard assembler syntax fields**.

\(<Rt>\) For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1, T2, T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\) For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>( U )</th>
<th>(+/-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

\(+\) Specifies the offset is added to the base register.

\(<imm>\) For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as \(<imm>/4\).

For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as \(<imm>/4\).

For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias (of variant)</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUSH</strong> (single register) A1 (pre-indexed)</td>
<td>( P == '1' ) &amp; ( U == '0' ) &amp; ( W == '1' ) &amp; ( Rn == '1101' ) &amp; ( imm12 == '000000000100' )</td>
</tr>
</tbody>
</table>
Alias | Of variant | Is preferred when
--- | --- | ---
PUSH (single register) | T4 (pre-indexed) | Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100'

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,4] = if t == 15 then PCStoreValue() else R[t];
    if wback then R[n] = offset_addr;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      offset_addr = if add then (B[n] + imm32) else (B[n] - imm32);
      address = if index then offset_addr else R[n];
      MemU[address,4] = R[t];
      if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 1 1</td>
</tr>
</tbody>
</table>

Offset (P == 1 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}}

Post-indexed (P == 0 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn], {+/-}<Rm>{, <shift>}}

Pre-indexed (P == 1 && W == 1)

STR{<c>}{<q>} <Rt>, [<Rn], {+/-}<Rm>{, <shift>}}!

if P == '0' && W == '1' then SEE "STRT";

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

if (shift_t, shift_n) = (SRType_LSL, 0);

CONstrained UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

T1

STR{<c>}{<q>} <Rt>, [<Rn], {+/-}<Rm>]

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

| t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); |
| index = TRUE; add = TRUE; wback = FALSE; |
| (shift_t, shift_n) = (SRType_LSL, 0); |
CONSTRANIED UNPREDICTABLE behavior

If \( t = 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRANIED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- **<c>** See Standard assembler syntax fields.
- **<q>** See Standard assembler syntax fields.
- **<Rt>** For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated. For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Rn>** For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.
- **+/-** Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

- **+** Specifies the index register is added to the base register.
- **<Rm>** Is the general-purpose index register, encoded in the "Rm" field.
- **<shift>** The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register.
- **<imm>** If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    if t == 15 then  // Only possible for encoding A1
        data = PCStoreValue();
    else
        data = R[t];
    MemU[address,4] = data;
    if wback then R[n] = offset_addr;

Operational information

If CPSR.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) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0  | 1  | 0  | P | U | 1 | W | 0  | Rn | Rt | imm12 |

cond

Offset (P == 1 && W == 0)

STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-.}<imm>}]

Post-indexed (P == 0 && W == 0)

STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-.<imm>

Pre-indexed (P == 1 && W == 1)

STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-.<imm>}

if P == '0' && W == '1' then SEE "STRBT";
if P == '0' && W == '1' then SEE "STRBT";

\[
\begin{align*}
\text{t} &= \text{UInt}(\text{Rt}); \\
\text{n} &= \text{UInt}(\text{Rn}); \\
\text{imm32} &= \text{ZeroExtend}(\text{imm12}, 32); \\
\text{index} &= (P == '1'); \\
\text{add} &= (U == '1'); \\
\text{wback} &= (P == '0') || (W == '1'); \\
\text{if} \ t &= 15 \text{ then UNPREDICTABLE}; \\
\text{if} \ wback \ && (n == 15 || n == t) \text{ then UNPREDICTABLE};
\end{align*}
\]

CONSTRAINED UNPREDICTABLE behavior

If \( t = 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>imm5</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
T1

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]  

{imm} = UInit(Rt);  {imm32} = ZeroExtend(imm5, 32);  
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | t != 1111 | Rt | imm12 |
---|----|-----|--------|
| 1 1 1 1 0 0 0 0 | = 1111 | 11 0 0 |
```

T2

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]  // (<Rt>, <Rn>, <imm> can be represented in T1)  

STRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]  

if Rn == '1111' then UNDEFINED;  
{imm} = UInit(Rt);  {imm32} = ZeroExtend(imm12, 32);  
index = TRUE;  add = TRUE;  wback = FALSE;  
if t == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

**CONSTRAINED UNPREDICTABLE behavior**

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | t != 1111 | Rt | 1 P U W | imm8 |
---|----|-----|-------|--------|
| 1 1 1 1 0 0 0 0 | = 1111 | 0 0 0 0 |
```

**Offset (P == 1 && U == 0 && W == 0)**

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #.<imm>}]  
```

**Post-indexed (P == 0 && W == 1)**

```
STRB{<c>}{<q>} <Rt>, [<Rn>], #.<.><imm>  
```

**Pre-indexed (P == 1 && W == 1)**

```
STRB{<c>}{<q>} <Rt>, [<Rn>, #<.+><imm>]  
```

if P == '1' && U == '1' && W == '0' then SEE "STRBT";  
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;  
{imm} = UInit(Rt);  {imm32} = ZeroExtend(imm8, 32);  
index = (P == '1');  add = (U == '1');  wback = (W == '1');  
if t == 15 || (wback && n == t) then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

**CONSTRAINED UNPREDICTABLE behavior**

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If \( \text{wback} \ & \ n == t \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If \( \text{wback} \ & \ n == 15 \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.
\(<Rn>\) For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.
For encoding T1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field.
\(+/-\) Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

\( +\) Specifies the offset is added to the base register.
\(<imm>\) For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.
For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the “imm5” field.
For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.
For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
      address = if index then offset_addr else R[n];
      MemU[address,1] = R[t]<7:0>;
      if wback then R[n] = offset_addr;
```
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 register to memory. The offset register value can optionally be shifted. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 1 | 1 | P | U | 1 | W | 0 | Rn | Rt | imm5 | stype | 0 | Rm |

cond

Offset (P == 1 && W == 0)

STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}}

Post-indexed (P == 0 && W == 0)

STRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Pre-indexed (P == 1 && W == 1)

STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}}!

if P == '0' && W == '1' then SEE "STRBT";

\[ t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \]

index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');

(shift_t, shift_n) = DecodeImmShift(stype, imm5);

if t == 15 || m == 15 then UNPREDICTABLE;

if wback && (n == 15 || n == t) then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | Rm | Rn | Rt |
T1

STRB{<c}]{<q} <Rt>, [<Rn>, {+}<Rm>]

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRTtype_LSL, 0);

T2

STRB{<c}>.W <Rt>, [<Rn>, {+<Rm}>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

STRB{<c}]{<q} <Rt>, [<Rn>, {+<Rm}>{, LSL #<imm>}]  

if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRTtype_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>  See Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<Rt>  Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn>  For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

/+/-  Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th></th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

+  Specifies the index register is added to the base register.

<Rm>  Is the general-purpose index register, encoded in the "Rm" field.

<shift>  The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

<imm>  If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STRBT

Store Register Byte Unprivileged stores a byte from a register to memory. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 1 0 0 U 1 1 0</td>
</tr>
</tbody>
</table>

cond

A1

STRBT{<c>}{<q>} {<Rt>, [<Rn>] } , `{, #/{-}imm}`

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');

register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);

if t == 15 || n == 15 || n == t then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If n == t, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If n == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
• The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 1 1 0 U 1 1 0</td>
</tr>
</tbody>
</table>
STRBT{<c>}{<q}> <Rt>, [<Rn>], {+/−}<Rm>{, <shift>}

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>1111</td>
<td></td>
<td>Rt</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T1**

STRBT{<c>}{<q}> <Rt>, [<Rn> {, #<imm>}]  

if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**CONSTRAINED UNPREDICTABLE behavior**

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>`  See Standard assembler syntax fields.
- `<q>`  See Standard assembler syntax fields.
For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

<\Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

<\Rm> Is the general-purpose index register, encoded in the "Rm" field.

<\shift> The shift to apply to the value read from <\Rm>. If absent, no shift is applied. Otherwise, see \textit{Shifts applied to a register}.

+ Specifies the offset is added to the base register.

<\imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

\textbf{Operation}

\begin{verbatim}
if \textbf{ConditionPassed}() then
  if PSTATE.EL == EL2 then UNPREDICTABLE;  // Hyp mode
  EncodingSpecificOperations();
  offset = if register form then \textbf{Shift}(R[m], shift_t, shift_n, PSTATE.C) else imm32;
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  address = if postindex then R[n] else offset_addr;
  \textbf{MemU\_unpriv}[address,1] = R[t]<7:0>;
  if postindex then R[n] = offset_addr;
\end{verbatim}

\textbf{CONstrained UNPREDICTABLE behavior}

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is \textbf{UNDEFINED}.
- The instruction executes as \textbf{NOP}.
- The instruction executes as \textbf{STRB} (immediate).

\textbf{Operational information}

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111| 0 | 0 | 0 | P | U | 1 | W | 0 | Rn | Rt | imm4H | 1 | 1 | 1 | 1 | imm4L |

 Offset (P == 1 && W == 0)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {}, #(+/-)<imm>]

Post-indexed (P == 0 && W == 0)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #(+/-)<imm>

Pre-indexed (P == 1 && W == 1)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #(+/-)<imm>]

if Rt<0> == '1' then UNPREDICTABLE:

if wback && (n == t || n == t2) then UNPREDICTABLE;
if wback && n == 15 then UNPREDICTABLE;

if Rt<0> == '1', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback && (n == t || n == t2), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If Rt<0> == '1', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes with the additional decode: t<0> = '0'.
• The instruction executes with the additional decode: t2 = t.
• The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

CONSTRAINED UNPREDICTABLE behavior
If $P == '0' \&\& W == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

### T1

$$
\begin{array}{cccccccccccccccccc}
  15 & 14 & 13 & 12 & 11 & 10 &  9 &  8 &  7 &  6 &  5 &  4 &  3 &  2 &  1 &  0 \\
\end{array}
$$

| 1 | 1 | 1 | 0 | 1 | 0 | P | U | 1 | W | 0 | != 1111 | Rt | Rt2 | imm8 |

Rn

**Offset (P == 1 \&\& W == 0)**

STRD{$<c>\{<q> \text{<Rt>}, \text{<Rt2>}, [\text{<Rn>} \{, \#\{+/-\}\text{<imm}>\}}$}

**Post-indexed (P == 0 \&\& W == 1)**

STRD{$<c>\{<q> \text{<Rt>}, \text{<Rt2>}, [\text{<Rn>}], \#\{+/-\}\text{<imm>}}$

**Pre-indexed (P == 1 \&\& W == 1)**

STRD{$<c>\{<q> \text{<Rt>}, \text{<Rt2>}, [\text{<Rn>}, \#\{+/-\}\text{<imm>}}$!}

If $P == '0' \&\& W == '0'$ then SEE "Related encodings";

```plaintext
if P == '1' then
  t = UInt(Rt); t2 = UInt(Rt2); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
  index = (P == '1'); add = (W == '1'); wback = (W == '1');
  if wback \&\& (n == t || n == t2) then UNPREDICTABLE;
  if n == 15 || t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

**CONSTRAINED UNPREDICTABLE behavior**

If $t == 15 || t2 == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback \&\& (n == t || n == t2), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If wback \&\& n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: *Load/store dual, load/store exclusive, table branch*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.
  
  For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2> For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>.
For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.
For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th></th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.

**Operation**

```cpp
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
        if BigEndian(AccType_ATOMIC) then
            data<63:32> = R[t];
            data<31:0> = R[t2];
        else
            data<31:0> = R[t];
            data<63:32> = R[t2];
    else
        MemA[address,8] = data;
        else
            MemA[address,4] = R[t];
            MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRD (register)

Store Register Dual (register) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

A1

```
<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>P</th>
<th>U</th>
<th>0</th>
<th>W</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>Rt</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Offset \((P == 1 && W == 0)\)

\[
\text{STRD}\{<c>\}\{<q>\} <Rt>, <Rt2>, [<Rn>, {+/−}<Rm>]
\]

Post-indexed \((P == 0 && W == 0)\)

\[
\text{STRD}\{<c>\}\{<q>\} <Rt>, <Rt2>, [<Rn>, {+/−}<Rm>]
\]

Pre-indexed \((P == 1 && W == 1)\)

\[
\text{STRD}\{<c>\}\{<q>\} <Rt>, <Rt2>, [<Rn>, {+/−}<Rm>]
\]

if \(Rt<0> == '1'\) then UNPREDICTABLE:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If \(wback && (n == t || n == t2)\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \(Rt<0> == '1'\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: \(t<0> = '0'\).
- The instruction executes with the additional decode: \(t2 = t\).
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when \(Rt == '1111'\).
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes with the additional decode: $P = '1'; W = '0'$.
• The instruction executes with the additional decode: $P = '1'; W = '1'$.
• The instruction executes with the additional decode: $P = '0'; W = '0'$.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<\texttt{c}>  See Standard assembler syntax fields.
<\texttt{q}>  See Standard assembler syntax fields.
<\texttt{Rt}> Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.
<\texttt{Rt2}> Is the second general-purpose register to be transferred. This register must be $<\texttt{R}(t+1)>$.
<\texttt{Rn}> Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.
$+/-$ Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "$U$":


<\texttt{Rm}> Is the general-purpose index register, encoded in the "Rm" field.

Operation

\begin{verbatim}
if \texttt{ConditionPassed}() then
  EncodingSpecificOperations();
  offset_addr = if add then (\texttt{R}[n] + \texttt{R}[m]) else (\texttt{R}[n] - \texttt{R}[m]);
  address = if index then offset_addr else \texttt{R}[n];
  if address == \texttt{Align}(address, 8) then
    bits(64) data;
    if \texttt{BigEndian}(\texttt{AccType_ATOMIC}) then
      data<63:32> = \texttt{R}[t];
      data<31:0> = \texttt{R}[t2];
    else
      data<31:0> = \texttt{R}[t];
      data<63:32> = \texttt{R}[t2];
    \texttt{MemA}[address,8] = data;
  else
    \texttt{MemA}[address,4] = \texttt{R}[t];
    \texttt{MemA}[address+4,4] = \texttt{R}[t2];
  if \texttt{wback} then \texttt{R}[n] = offset_addr;
\end{verbatim}

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, stores a word from a register to the calculated address if the PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

STREX{<c>{<q>}}<Rd>, <Rt>, [{<Rn>} {, #} <imm>]

d = UInt(Rd); t = UInt(Rt); n = UInt(Rn); imm32 = Zeors(32); // Zero offset
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

|15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|   | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

T1

STREX{<c>{<q>}}<Rd>, <Rt>, [{<Rn>} {, #} <imm>]

d = UInt(Rd); t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRUINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:
   0  If the operation updates memory.
   1  If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<imm> For encoding A1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted.
       For encoding T1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:
   • Memory is not updated.
   • <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
   • If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
   • Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.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

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STREXB

Store Register Exclusive Byte derives an address from a base register value, stores a byte from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

cond

A1

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 0 0 1 1 0 0</td>
</tr>
</tbody>
</table>

T1

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0   If the operation updates memory.

1   If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

• Memory is not updated.
• <Rd> is not updated.

If AArch32.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

if ConditionPassed() then
   EncodingSpecificOperations();
   address = R[n];
   if AArch32.ExclusiveMonitorsPass(address,1) then
      MemA[address,1] = R[t]<7:0>;
      R[d] = ZeroExtend('0');
   else
      R[d] = ZeroExtend('1');

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STREXD

Store Register Exclusive Doubleword derives an address from a base register value, stores a 64-bit doubleword from two registers to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see Synchronization and semaphores. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Rn | Rd | (1)(1) | 1 | 1 | 1 | 0 | 0 | 1 | Rt |

cond

A1

STREXD{<c>{<q>}}<Rd>, <Rt>, <Rt2>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);
if d == 15 || Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d == t, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs the store to an UNKNOWN address.

If Rt<0> == '1', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes with the additional decode: Rt<0> = '0'.
• The instruction executes with the additional decode: t2 = t.
• The instruction executes as described, with no change to its behavior and no additional side effects.

If Rt == '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction is handled as described in Using R15.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Rn | Rd | Rt | Rt2 | 0 | 1 | 1 | 1 | Rd |
T1

STREXD\{<c>\}\{<q>\} <Rd>, <Rt>, <Rt2>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t || d == t2 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:
0  If the operation updates memory.
1  If the operation fails to update memory.

<Rd> must not be the same as <Rn>, <Rt>, or <Rt2>.

<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.
For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

<Rt2> For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.
For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn> Is the general-purpose base register, 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.
- <Rd> is not updated.

A non doubleword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.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

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian(AccType_ATOMIC) then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address,8) then
        MemA[address,8] = value;  R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**STREXH**

Store Register Exclusive Halfword derives an address from a base register value, stores a halfword from a register to
the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value
of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores*. For information about
memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| !=1111 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rn | Rd | (1)(1) | 1 | 1 | 1 | 0 | 0 | 1 | Rt |

cond

**A1**

STREXH{<c>{<q>}} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Rn | Rt | (1)(1)(1)(1) | 0 | 1 | 0 | 1 | Rd |

**T1**

STREXH{<c>{<q>}} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The instruction performs the store to an **UNKNOWN** address.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:
  - 0 If the operation updates memory.
  - 1 If the operation fails to update memory.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, 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.
- `<Rd>` 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 AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is **IMPLEMENTATION DEFINED** whether the exception is generated.

If AArch32.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**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t]<15:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | P | U | 1 | W | 0 | Rn | Rt | imm4H | 1 | 0 | 1 | 1 | imm4L |
| cond |

Offset (P == 1 & W == 0)

STRH{<c>}{<q>}{<Rt>}, [<Rn> {, #/{+/-}<imm}>]

Post-indexed (P == 0 & W == 0)

STRH{<c>}{<q>}{<Rt>}, [<Rn>], #{+/-.}<imm>

Pre-indexed (P == 1 & W == 1)

STRH{<c>}{<q>}{<Rt>}, [<Rn>, #{+/-.}<imm>]

if P == '0' & W == '1' then SEE "STRHT";

t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && n == t then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback & n == t, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If wback & n == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 0 | 0 | 0 | 0 | imm5 | Rn | Rt |
T1

STRH{<c} {<q} <Rt>, [<Rn> {, #}{<imm}>]}

t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE; add = TRUE; wback = FALSE;

T2

if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = TRUE; add = TRUE; wback = FALSE;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRANGED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

* The instruction is UNDEFINED.
* The instruction executes as NOP.
* The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3

if P == '1' && U == '1' && W == '0' then SEE "STRHT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
index = (P == '1'); add = (U == '1'); wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRANGED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

* The instruction is UNDEFINED.
* The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to 
R15 is UNKNOWN.

If \( \text{wback} \&\& \ n == t \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\) For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in 
the offset variant, but this is deprecated.

For encoding A1, T1, T2, T3: is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and 
encoded in "U":

<table>
<thead>
<tr>
<th></th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

+ Specifies the offset is added to the base register.

\(<\text{imm}>\) For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if 
omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 
to 62, defaulting to 0 and encoded in the "imm5" field as \(<\text{imm}>/2\).

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting 
to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if 
omitted, and encoded in the "imm8" field.

Operation

if \( \text{CurrentInstrSet}() == \text{InstrSet_A32} \) then
    if \( \text{ConditionPassed}() \) then
        EncodingSpecificOperations();
        offset_addr = if add then (\(R[n]\) + imm32) else (\(R[n]\) - imm32);
        address = if index then offset_addr else \(R[n]\);
        \(\text{MemU}[address,2] = R[t]<15:0>\);
        if \( \text{wback} \) then \(R[n] = offset_addr\);
    else
        if \( \text{ConditionPassed}() \) then
            EncodingSpecificOperations();
            offset_addr = if add then (\(R[n]\) + imm32) else (\(R[n]\) - imm32);
            address = if index then offset_addr else \(R[n]\);
            \(\text{MemU}[address,2] = R[t]<15:0>\);
            if \( \text{wback} \) then \(R[n] = offset_addr\);
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and
stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For
information about memory accesses see Memory accesses.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=| 1111| 0  | 0  | 0  | P | U | 0 | W | 0  | Rn | Rt | (0) | (0) | (0) | (0) | 1 | 0 | 1 | 1  | Rm |
```

cond

**Offset** (P == 1 && W == 0)

```
STRH{<c>}{<q>}{<Rt>}, [<Rn>, {+/-}<Rm>]
```

**Post-indexed** (P == 0 && W == 0)

```
STRH{<c>}{<q>}{<Rt>}, [<Rn>], {+/-}<Rm>
```

**Pre-indexed** (P == 1 && W == 1)

```
STRH{<c>}{<q>}{<Rt>}, [<Rn>, {+/-}<Rm>]
```

if P == '0' && W == '1' then SEE "STRHT";

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRTType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && n == t then UNPREDICTABLE;
if wback && n == 15 then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

```
<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The instruction `STRH<c>{<q>} <Rt>, [<Rn>, {+}<Rm>]` can be represented as:

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTypre LSL, 0);
```

### T2

```
| 1 0 1 1 1 1 0 0 0 0 0 1 0 | !=1111 | Rt | 0 0 0 0 0 0 0 | imm2 | Rm |
```

The instruction `STRH<c>{<q>} {W} <Rt>, [<Rn>, {+}<Rm>]` can be represented in T1.

```
STRH<c>{<q>} <Rt>, [<Rn>, {+}<Rm>] {, LSL #<imm>}
```

### CONSTRANIED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRANIED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the “Rt” field.
- `<Rn>` For encoding A1: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.
- `+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:
  
  | 0 | +/
  |---|---
  | 1 | -

- `+` Specifies the index register is added to the base register.
- `<Rm>` Is the general-purpose index register, encoded in the “Rm” field.
- `<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;
    if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRHT

Store Register Halfword Unprivileged stores a halfword from a register to memory. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|-------|------------------|------------------|
| != 1111 | 0 0 0 | 0 U 1 1 0 | Rn | Rt | imm4H | 1 0 1 1 | imm4L |
| cond |

A1

STRHT{<c>}{<q>}{<Rt>}, [<Rn>] {, #{+/−}<imm>}

t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|-------|------------------|------------------|
| != 1111 | 0 0 0 | 0 U 0 1 0 | Rn | Rt | (0)(0)(0)(0) | 1 0 1 1 | Rm |
| cond |
A2

\[
\text{STRHT}\{<c>\}\{<q>\} \ {<Rt>}, \ [<Rn>\}, \ {+/-}\{<imm>\}
\]

\[
t = \text{U}n\text{i}nt(Rt); \ n = \text{U}n\text{i}nt(Rn); \ m = \text{U}n\text{i}nt(Rm); \ \text{postindex} = \text{TRUE}; \ \text{add} = (U == '1'); \\
\text{register form} = \text{TRUE}; \\
\text{if } t == 15 \text{ || } n == 15 \text{ || } n == t \text{ || } m == 15 \text{ then UNPREDICTABLE;}
\]

CONSTRAINED UNPREDICTABLE behavior

If \( t == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If \( n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( n == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/ For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

**Operation**

if ConditionPassed() then
  if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
  EncodingSpecificOperations();
  offset = if register_form then R[m] else imm32;
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  address = if postindex then R[n] else offset_addr;
  MemU_unpriv[address,2] = R[t]<15:0>;
  if postindex then R[n] = offset_addr;

**CONSTRAINED UNPREDICTABLE behavior**

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as STRH (immediate).

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STRT

Store Register Unprivileged stores a word from a register to memory. For information about memory accesses see Memory accesses.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------|-----------|
| != 1111 | 0 1 0 0 | U 0 1 0 | Rn | Rt | imm12 |
| cond |

A1

STRT{<c>}{<q>} <Rt>, [<Rn>] , #/<+/-><imm>

t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');

if n == 15 || n == t then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------|-----------|
| != 1111 | 0 1 1 0 | U 0 1 0 | Rn | Rt | imm5 | stype | 0 | Rm |
| cond |

A2

STRT{<c>}{<q>} <Rt>, [<Rn>], {+/+}<Rm>{, <shift}>

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');

if n == 15 || n == t || m == 15 then UNPREDICTABLE;
CONstrained UNPREDICTABLE behavior

If \( n = t \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \( n = 15 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in **Using R15**.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | Rn |
| 1 1 1 1 0 0 0 1 0 0 | != 1111 | Rt | 1 1 1 0 | imm8 |

T1

\[
\text{STRT}{\{c\}}{\{q\}}\ <Rt>, \ [<Rn> \ {, \ #\{+\}<imm>}]\]

if \( Rn \) == '1111' then **UNDEFINED**;

\[
t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \ postindex = \text{FALSE}; \ add = \text{TRUE};\]

\[
\text{register} \_\text{form} = \text{FALSE}; \ \text{imm32} = \text{ZeroExtend}(\text{imm8}, \ 32);\]

if \( t \) == 15 then **UNPREDICTABLE**; // Armv8-A removes **UNPREDICTABLE** for R13

CONstrained UNPREDICTABLE behavior

If \( t = 15 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is **UNKNOWN**.

For more information about the **CONstrained UNPREDICTABLE** behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\)

See Standard assembler syntax fields.

\(<q>\)

See Standard assembler syntax fields.

\(<Rt>\)

For encoding A1 and A2: is the general-purpose register to be transferred, encoded in the "Rt" field.

The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\)

Is the general-purpose base register, encoded in the "Rn" field.

\/+\)

For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

\[
\begin{array}{c|c}
\text{U} & +/- \\
0 & - \\
1 & + \\
\end{array}
\]

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

STRT
<Rm>  Is the general-purpose index register, encoded in the "Rm" field.

<shift>  The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register.

+  Specifies the offset is added to the base register.

<imm>  For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

if ConditionPassed() then
  if PSTATE.EL == EL2 then UNPREDICTABLE;       // Hyp mode
  EncodingSpecificOperations();
  offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  address = if postindex then R[n] else offset_addr;
  if t == 15 then  // Only possible for encodings A1 and A2
    data = PCStoreValue();
  else
    data = R[t];
    MemU_unpriv[address,4] = data;
  if postindex then R[n] = offset_addr;

CONSTRUANED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as STR (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**SUB (immediate, from PC)**

Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This is an alias of **ADR**. This means:

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

It has encodings from the following instruction sets: A32 ( **A2** ) and T32 ( **T2** ).

### A2

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | Rd | imm12 |

**cond**

**A2**

\[
\text{SUB}\{<c}\}{<q}\} <Rd>, PC, \#<const>
\]

is equivalent to

**ADR\{<c}\}{<q}\} <Rd>, <label>**

and is the preferred disassembly when \(\text{imm12} = '000000000000'\).

### T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>i</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>imm3</td>
</tr>
</tbody>
</table>

**T2**

\[
\text{SUB}\{<c}\}{<q}\} <Rd>, PC, \#<imm12>
\]

is equivalent to

**ADR\{<c}\}{<q}\} <Rd>, <label>**

and is the preferred disassembly when \(i:imm3:imm8 = '000000000000'\).

### Assembler Symbols

- **<c>** See *Standard assembler syntax fields*.
- **<q>** See *Standard assembler syntax fields*.
- **<Rd>** For encoding A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
  
  For encoding T2: is the general-purpose destination register, encoded in the "Rd" field.

- **<label>** For encoding A2: the label of an instruction or literal data item whose address is to be loaded into \(<Rd>\). The assembler calculates the required value of the offset from the Align(PC, 4) value of the **ADR** instruction to this label.
  
  If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset.
  
  If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32.
  
  Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions*.
For encoding T2: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.

If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.
If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32.
Permitted values of the size of the offset are 0-4095.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
<const> An immediate value. See Modified immediate constants in A32 instructions for the range of values.

Operation

The description of ADR gives the operational pseudocode for this instruction.
SUB, SUBS (immediate)

Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SUBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode, except for encoding T5 with <imm8> set to zero, which is the encoding for the ERET instruction, see *ERET*.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3, T4 and T5).

### A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
!= 1111 0 0 1 0 0 1 0 5 | Rn | Rd | imm12
cond
```

**SUB (S == 0 && Rn != 11x1)**

```
SUB{<c>}{<q>}{<Rd>,}{<Rn>,}{#<const>
```

**SUBS (S == 1 && Rn != 1101)**

```
SUBS{<c>}{<q>}{<Rd>,}{<Rn>,}{#<const>
```

```
if Rn == '1111' && S == '0' then SEE "ADR";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

### T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 1 | imm3 | Rn | Rd
```

### T1

```
SUB{<q>{<Rd>,}<Rn>,}{#<imm3}> (Inside IT block)
```

```
SUBS{<q>{<Rd>,}<Rn>,}{#<imm3}> (Outside IT block)
```

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

### T2

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 1 | Rdn | imm8
```
SUB<q> <Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> can be represented in T1)
SUB<q> {<Rdn>}, <Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)
SUBS{<q>} <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)
SUBS{<q>} {<Rdn>}, <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rdn); \quad \text{setflags} = !\text{InITBlock}(); \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32);
\]

T3

\[
1 1 1 1 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 = 1101 0 0 \quad \text{imm32} \quad \text{Rd} \quad \text{imm8}
\]

SUB (S == 0)

SUB<.W {<Rd>}, <Rdn>, #<const> // (Inside IT block, and <Rd>, <Rdn>, <const> can be represented in T1 or T2)
SUB<>{<q>} {<Rd>}, <Rdn>, #<const>

SUBS (S == 1 && Rd != 1111)

SUBS.W {<Rd>}, <Rdn>, #<const> // (Outside IT block, and <Rd>, <Rdn>, <const> can be represented in T1 or T2)
SUBS<>{<q>} {<Rd>}, <Rdn>, #<const>

if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd); \quad n = UInt(Rdn); \quad \text{setflags} = (S == '1'); \quad \text{imm32} = \text{T32ExpandImm}(i:imm3:imm8);
if (d == 15 || !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T4

\[
1 1 1 1 0 1 0 1 0 1 0 1 1 0 1 1 0 = 11x1 0 0 \quad \text{imm32} \quad \text{Rd} \quad \text{imm8}
\]

T4

SUB<>{<q>} {<Rd>}, <Rdn>, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)
SUBW<>{<q>} {<Rd>}, <Rdn>, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

if Rn == '1111' then SEE "ADR";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd); \quad n = UInt(Rdn); \quad \text{setflags} = \text{FALSE}; \quad \text{imm32} = \text{ZeroExtend}(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T5

\[
1 1 1 1 0 0 1 1 1 1 0 1 1 1 1 1 1 0 = 00000000 
\]

SUB, SUBS (immediate)
**T5**

**SUBS{<c>}{<q>} PC, LR, #<imm8>**

if Rn == '1110' && IsZero(imm8) then SEE "ERET";
if n == 15 then UInt(Rn); setflags = TRUE; imm32 = ZeroExtend(imm8, 32);
if n != 14 then UNPREDICTABLE;
if InitInITBlock() && !LastInITBlock() then UNPREDICTABLE;

For more information about the **CONstrained UNPREDICtable** behavior of this instruction, see *Architectural Constraints on UNPREDICtable behaviors*, and particularly **SUBS PC, LR and related instructions (A32)** and **SUBS PC, LR and related instructions (T32)**.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rdn>` Is the general-purpose source and destination register, encoded in the "Rdn" field.
- `<imm8>` For encoding T2: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. If the PC is used:
  - For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
  - For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction unless `<Rn>` is the LR.
- `<Rn>` For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see **SUB (SP minus immediate)**. If the PC is used, see **ADR**.
- `<imm3>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.
- `<imm12>` Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
- `<const>` For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.
  - For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

In the T32 instruction set, MOV{<c>}{<q>} PC, LR is a pseudo-instruction for SUBS{<c>}{<q>} PC, LR, #0.

**Operation**

if ConditionPassed() then
  EncodingSpecificOperations();
  (result, nzcv) = AddWithCarry(R[n], NOT(imm32), '1');
  if d == 15 then
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
  else
    R[d] = result;
    if setflags then
      PSTATE.<N,Z,C,V> = nzcv;
Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

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

Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 5 | != | 1101 | Rd | imm5 | stype | 0 | Rm |
| cond | |

**SUB, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)**

```
SUB{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, RRX
```

**SUB, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))**

```
SUB{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**SUBS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)**

```
SUBS{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, RRX
```

**SUBS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))**

```
SUBS{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

if Rn == '1101' then SEE "SUB (SP minus register)"

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**T1**

```
SUB{<c>}{<q>}{<Rd>, <Rn>, <Rm> // (Inside IT block)
SUBS{<c>}{<Rd>,} <Rn>, <Rm> // (Outside IT block)
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```
T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

\[
Rn = 11101 \quad (0) \quad \text{imm3} \quad \text{Rd} \quad \text{imm2} \quad \text{stype} \quad \text{Rm}
\]

SUB, rotate right with extend \((S == 0 && \text{imm3} == 000 && \text{imm2} == 00 && \text{stype} == 11)\)

\[
\text{SUB}\{<c>\}\{<q>\} \{<Rd>, \} \{<Rn>, <Rm>, \} \text{RRX}
\]

SUB, shift or rotate by value \((S == 0 && !(\text{imm3} == 000 && \text{imm2} == 00 && \text{stype} == 11))\)

\[
\text{SUB<c>.W} \{<Rd>,\} \{<Rn>, <Rm>\} /\ (\text{Inside IT block, and } <Rd>, <Rn>, <Rm> \text{ can be represented in T1})
\]

\[
\text{SUB}\{<c>\}\{<q>\} \{<Rd>,\} \{<Rn>, <Rm>\} \{, <shift> \#<amount>\}
\]

SUBS, rotate right with extend \((S == 1 && \text{imm3} == 000 && \text{Rd} != 1111 && \text{imm2} == 00 && \text{stype} == 11)\)

\[
\text{SUBS}\{<c>\}\{<q>\} \{<Rd>, \} \{<Rn>, <Rm>, \} \text{RRX}
\]

SUBS, shift or rotate by value \((S == 1 && !(\text{imm3} == 000 && \text{imm2} == 00 && \text{stype} == 11)) && \text{Rd} != 1111\)

\[
\text{SUBS.W} \{<Rd>,\} \{<Rn>, <Rm>\} /\ (\text{Outside IT block, and } <Rd>, <Rn>, <Rm> \text{ can be represented in T1})
\]

\[
\text{SUBS}\{<c>\}\{<q>\} \{<Rd>,\} \{<Rn>, <Rm>\} \{, <shift> \#<amount>\}
\]

if Rd == '1111' && S == '1' then SEE "CMP (register)"
if Rn == '1101' then SEE "SUB (SP minus register)"
if Rm == '11101' then SEE "SUB (SP minus register)"
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift t, shift n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\quad \text{See Standard assembler syntax fields.}\)
\(<q>\quad \text{See Standard assembler syntax fields.}\)
\(<Rd>\quad \text{For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as } <Rn>. \text{Arm deprecates using the PC as the destination register, but if the PC is used:}\)
\begin{itemize}
  \item For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
  \item For the SUBS variant, the instruction performs an exception return, that restores \text{PSTATE} from SPSR_<current_mode>.
\end{itemize}

\(<Rn>\quad \text{For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. If the SP is used, see } \text{SUB (SP minus register)}.\)

\(<Rm>\quad \text{For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.}\)

\(<\text{shift}>\quad \text{Is the type of shift to be applied to the second source register, encoded in "stype".}\)
<table>
<thead>
<tr>
<th>stype</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

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

Subtract (register-shifted register) subtracts a register-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------------------------|----------|----------|----------|----------|----------|----------|----------|
| != 1111 0 0 0 0 0 1 0 S Rn Rd Rs 0 stype 1 Rm |

cond

Flag setting (S == 1)

SUBS{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, <shift> <Rs>

Not flag setting (S == 0)

SUB{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, <shift> <Rs>

d =.UInt(Rd); n = .UInt(Rn); m = .UInt(Rm); s = .UInt(Rs);
setflags = (S == '1'); shift_t = .DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in "stype":

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

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  shift_n = .UInt(R[s]<7:0>);
  shifted = .Shift(R[m], shift_t, shift_n, PSTATE.C);
  (result, nzcv) = .AddWithCarry(R[n], NOT(shifted), '1');
  R[d] = result;
  if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.


**SUB, SUBS (SP minus immediate)**

Subtract from SP (immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores `PSTATE` from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2 and T3).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>!= 1111</th>
<th>0 0 1 0</th>
<th>0 1 0</th>
<th>5</th>
<th>1</th>
<th>1 0 1</th>
<th>Rd</th>
<th>imm12</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SUB (S == 0)**

```
SUB{{}<q}> {<Rd>,} SP, #<const>
```

**SUBS (S == 1)**

```
SUBS{{}<q}> {<Rd>,} SP, #<const>
```

\[
d = \text{UInt}(Rd); \quad \text{setflags} = (S == '1'); \quad \text{imm32} = \text{A32ExpandImm}(\text{imm12});
\]

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>1 0 1 1 0 0 0 0 1</th>
<th>imm7</th>
</tr>
</thead>
</table>

**T1**

```
SUB{{}<q}> {SP,} SP, #<imm7>
```

\[
d = 13; \quad \text{setflags} = \text{FALSE}; \quad \text{imm32} = \text{ZeroExtend}(\text{imm7}:'00', 32);
\]

### T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 0 | i | 0 1 | 1 0 1 | S | 1 | 1 0 1 | 0 | imm3 | Rd | imm8 |
|----------------------------------------|----------|---|-----|-----|---|----|-----|-----|--------|-----|-------|-----|--------|

SUB, SUBS (SP minus immediate)
SUB (S == 0)

SUB{<c>}.W {<Rd>,} SP, #=><const> // (<Rd>, <const> can be represented in T1)

SUB{<c>}{<q>}{<Rd>,} SP, #=><const>

SUBS (S == 1 & Rd != 1111)

SUBS{<c>}{<q>}{<Rd>,} SP, #=><const>

if Rd == '1111' & S == '1' then SEE "CMP ( immediate)";
d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 & !setflags then UNPREDICTABLE;

T3

| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|------------------------|--------|--------|
| 1 | 1 | 1 | 1 | 1 | 0 | i | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | imm3 | Rd | imm8 |

T3

SUB{<c>}{<q>}{<Rd>,} SP, #=><imm12> // (<imm12> cannot be represented in T1, T2, or T3)

SUBW{<c>}{<q>}{<Rd>,} SP, #=><imm12> // (<imm12> can be represented in T1, T2, or T3)

d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

SP, Is the stack pointer.

<imm7> Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. If the PC is used:

• For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

• For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction unless <Rn> is the LR. For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T2: an immediate value. See Modified immediate constants in T32 instructions for the range of values.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(SP, NOT(imm32), '1');
    if d == 15 then  // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
**SUB, SUBS (SP minus register)**

Subtract from SP (register) subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores `PSTATE` from `SPSR_<current_mode>`.
  - The PE checks `SPSR_<current_mode>` for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|-----------------|---------------|---------------|
| != 1111                                      | 0 0 0 0 0 0 0 0 | 5 1 1 0 1 1 0 1 | Rs             | imm5          | stype 0       | Rm             |
| cond                                         |                 |                 |               |               |

**SUB, rotate right with extend (S == 0 & imm5 == 00000 & stype == 11)**

\[
\text{SUB\{c\}\{q\} \{<Rd>,\} \text{ SP, <Rm> }, \text{ RRX}}
\]

**SUB, shift or rotate by value (S == 0 & !(imm5 == 00000 & stype == 11))**

\[
\text{SUB\{c\}\{q\} \{<Rd>,\} \text{ SP, <Rm} \{, <\text{shift} \#<amount>}}
\]

**SUBS, rotate right with extend (S == 1 & imm5 == 00000 & stype == 11)**

\[
\text{SUBS\{c\}\{q\} \{<Rd>,\} \text{ SP, <Rm> }, \text{ RRX}}
\]

**SUBS, shift or rotate by value (S == 1 & !(imm5 == 00000 & stype == 11))**

\[
\text{SUBS\{c\}\{q\} \{<Rd>,\} \text{ SP, <Rm} \{, <\text{shift} \#<amount>}}
\]

\[
d = \text{UInt}(Rd); \ m = \text{UInt}(Rm); \ \text{setflags} = (S == '1');
\]

\[
\text{(shift_t, shift_n) = DecodeImmShift(stype, imm5)};
\]

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------|----------------|----------------|---------------|
| 1 1 1 0 1 1 1 0 1 1 1 0 1 1 0 1 | (0) | imm3 | Rd | imm2 | stype | Rm |
SUB, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

SUB{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX

SUB, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SUB{<c>}{<q>} {<Rd>,} SP, <Rm> // (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SUBS, rotate right with extend (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

SUBS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX

SUBS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in "stype":

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

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(SP, NOT(shifted), '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
SVC

Supervisor Call causes a Supervisor Call exception. For more information, see Supervisor Call (SVC) exception. SVC was previously called SWI, Software Interrupt, and this name is still found in some documentation.

Software can use this instruction as a call to an operating system to provide a service.

In the following cases, the Supervisor Call exception generated by the SVC instruction is taken to Hyp mode:

- If the SVC is executed in Hyp mode.
- If HCR.TGE is set to 1, and the SVC is executed in Non-secure User mode. For more information, see Supervisor Call exception, when HCR.TGE is set to 1

In these cases, the HSR, Hyp Syndrome Register identifies that the exception entry was caused by a Supervisor Call exception, EC value 0x11, see Use of the HSR. The immediate field in the HSR:

- If the SVC is unconditional:
  - For the T32 instruction, is the zero-extended value of the imm8 field.
  - For the A32 instruction, is the least-significant 16 bits the imm24 field.
- If the SVC is conditional, is UNKNOWN.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccccccc}
\hline
1 & 1 & 1 & 1 & \text{cond} & \text{imm24} \\
\end{array}
\]

A1

\[
\text{SVC}\{<c>}\{<q>\} \{\text{#}\}<\text{imm}> \\
\text{imm32} = \text{ZeroExtend}(\text{imm24}, 32);
\]

T1

\[
\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 1 & \text{imm8} \\
\end{array}
\]

T1

\[
\text{SVC}\{<c>\}{<q>\} \{\text{#}\}<\text{imm}> \\
\text{imm32} = \text{ZeroExtend}(\text{imm8}, 32);
\]

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<\text{imm}>\) For encoding A1: is a 24-bit unsigned immediate, in the range 0 to 16777215, encoded in the “imm24” field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm24 in software, for example to determine the required service.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm8 in software, for example to determine the required service.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  AArch32.CheckForSVCTrap(imm32<15:0>);
  AArch32.CallSupervisor(imm32<15:0>);
SXTAB

Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
|!= 1111| 0 1 1 0 1 0 1 0 |!= 1111| Rd  rotate(0)[0] 0 1 1 1 | Rm
```

cond  
Rn

A1

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

if Rn == '1111' then SEE "SXTB";
    d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
|1 1 1 1 0 1 0 0|1 0 0|!= 1111| 1 1 1 1 | Rd 1[0]rotate | Rm
```

Rn

T1

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

if Rn == '1111' then SEE "SXTB";
    d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<amount>` Is the rotate amount, encoded in "rotate":

<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Operation

if `ConditionPassed()` then
    EncodingSpecificOperations();
    rotated = `ROR(R[m], rotation);
    R[d] = R[n] + `SignExtend(rotated<7:0>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SXTAB16

Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| != 1111 | 0 1 1 0 1 0 0 | != 1111 | Rd | rotate(0)|[0] | 0 1 1 1 | Rm |
| cond | Rd | rotate(0) | Rm |

A1

SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

if Rn == '1111' then SEE "SXTB16";
if d == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 1 1 0 1 0 0 | != 1111 | 1 1 1 1 | Rd | 1 |(0)|rotate | Rm |
| Rn |

T1

SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

if Rn == '1111' then SEE "SXTB16";
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in "rotate":

<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>
Operation

```c
if (ConditionPassed()) then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0> = R[n]<15:0> + SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| !|= 1111 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | !|= 1111 | Rd | rotate[0](0) | 0 | 1 | 1 | 1 | Rm |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | Rd | rotate[0](0) | 0 | 1 | 1 | 1 | Rm |

### T1

| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | !|= 1111 | 1 | 1 | 1 | 1 | Rd | 1 | (0) | rotate | Rm |
| Rn | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | !|= 1111 | 1 | 1 | 1 | 1 | Rd | 1 | (0) | rotate | Rm |

### Assembler Symbols

- **<c>** See [Standard assembler syntax fields](#).
- **<q>** See [Standard assembler syntax fields](#).
- **<Rd>** Is the general-purpose destination register, encoded in the "Rd" field.
- **<Rn>** Is the first general-purpose source register, encoded in the "Rn" field.
- **<Rm>** Is the second general-purpose source register, encoded in the "Rm" field.
- **<amount>** Is the rotate amount, encoded in "rotate":

<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

For more information about the constrained UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<15:0>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>![1111]</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td>rotate</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**cond**

\begin{verbatim}
SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
\end{verbatim}

\begin{verbatim}
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
\end{verbatim}

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | Rm | Rd |

### T2

\begin{verbatim}
SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
\end{verbatim}

\begin{verbatim}
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
\end{verbatim}

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>`: See Standard assembler syntax fields.
- `<q>`: See Standard assembler syntax fields.
- `<Rd>`: Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>`: Is the general-purpose source register, encoded in the "Rm" field.
- `<amount>`: Is the rotate amount, encoded in "rotate";
Operation

if \text{ConditionPassed}() \text{ then}
    \text{EncodingSpecificOperations();}
    \text{rotated = ROR(R[m], rotation);}
    R[d] = \text{SignExtend}(\text{rotated}<7:0>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SXTB16

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| ! != 1111 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Rd | rotate(0)(0) | 0 | 1 | 1 | 1 | Rm |

cond

A1

\[
\text{SXTB16} \{<c>\} \{<q>\} \{<Rd>,\} \{<Rm>\} \{(ROR \ #<amount>)
\]

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate}:'000');
\]

if \( d == 15 \) || \( m == 15 \) then UNPREDICTABLE;

T1

| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | Rd | 1 | [0]rotate | Rm |

T1

\[
\text{SXTB16} \{<c>\} \{<q>\} \{<Rd>,\} \{<Rm>\} \{(ROR \ #<amount>)
\]

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate}:'000');
\]

if \( d == 15 \) || \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rm>\) Is the general-purpose source register, encoded in the "Rm" field.

\(<amount>\) Is the rotate amount, encoded in "rotate":

\[
\begin{array}{|c|c|}
\hline
\text{rotate} & \text{<amount>} \\
\hline
00 & (omitted) \\
01 & 8 \\
10 & 16 \\
11 & 24 \\
\hline
\end{array}
\]

Operation

if \( \text{ConditionPassed}() \) then

EncodingSpecificOperations();

\[
\text{rotated} = \text{ROR}(R[m], \text{rotation});
\]

\[
R[d]<15:0> = \text{SignExtend}((\text{rotated}<7:0>, 16);
\]

\[
R[d]<31:16> = \text{SignExtend}((\text{rotated}<23:16>, 16);
\]
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

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

Signed Extend Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--------|--------|
| != 1111                                       | 0 1 1 0 1 0 1 1 1 1 1 1 | Rd     | rotate | 0 1 1 1 | Rm     |

cond

A1

SXTH{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}

d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|--------|
| 1 0 1 1 0 0 1 0 0 0 1 0 1 0 0 Rm  Rd    |

T1

SXTH{<c>}{<q>} {<Rd>}, <Rm>

d = UInt(Rd); m = UInt(Rm); rotation = 0;

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|--------|
| 1 1 1 1 1 0 1 0 0 0 0 1 1 1 1 1 1 1 1 | Rd     | rotate | 1 0 | Rm    |

T2

SXTH{<c>}.W {<Rd>}, <Rm> // (<Rd>, <Rm> can be represented in T1)

SXTH{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}

d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c>     See Standard assembler syntax fields.

<q>     See Standard assembler syntax fields.

<Rd>    Is the general-purpose destination register, encoded in the "Rd" field.

<Rm>    Is the general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”: 

<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

**Operation**

```java
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Table Branch Byte or Halfword causes a PC-relative forward branch using a table of single byte or halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value returned from the table.

T1

| 1 1 1 0 1 0 0 0 1 1 0 1 | Rn | (1)|(1)|(1)|(1)|(0)|(0)|(0)|(0) | 0 0 0 | H | Rm |

Byte (H == 0)

TBB{<c>}{<q>} [<Rn>, <Rm>] // (Outside or last in IT block)

Halfword (H == 1)

TBH{<c>}{<q>} [<Rn>, <Rm>, LSL #1] // (Outside or last in IT block)

n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if InIBlock() && !LastInIBlock() then UNPREDICTABLE;

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<Rn>
Is the general-purpose base register holding the address of the table of branch lengths, encoded in the "Rn" field. The PC can be used. If it is, the table immediately follows this instruction.

<Rm>
For the byte variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.

For the halfword variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC + 2*halfwords, BranchType_INDIR);
**TEQ (immediate)**

Test Equivalence (immediate) performs a bitwise exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111| 0  | 0  | 1  | 0  | 1  | 1  | Rn | [0]| [0]| [0]| [0]| [0]| [0]| imm12 |

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>i</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>0</td>
<td>imm3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

< c > For encoding A1: the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

< q > For encoding T1: the general-purpose source register, encoded in the "Rn" field.

< Rn > For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

< const > For encoding A1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

**Operation**

if ConditionPassed() then
   EncodingSpecificOperations();
   result = R[n] EOR imm32;
   PSTATE.N = result<31>;
   PSTATE.Z = IsZeroBit(result);
   PSTATE.C = carry;
   // PSTATE.V unchanged
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Test Equivalence (register) performs a bitwise exclusive OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | Rn | [0][0][0][0] | imm5 | stype | 0 | Rm |

cond

Rotate right with extend (imm5 == 00000 && stype == 11)

TEQ{<c>}{<q>} <Rn>, <Rm>, RRX

Shift or rotate by value (!((imm5 == 00000 && stype == 11))

TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift>#<amount>}

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | Rn | [0] | imm3 | 1 | 1 | 1 | 1 | imm2 | stype | Rm |

Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

TEQ{<c>}{<q>} <Rn>, <Rm>, RRX

Shift or rotate by value (!((imm3 == 000 && imm2 == 00 && stype == 11))

TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift>#<amount>}

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);

if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:
<table>
<thead>
<tr>
<th>stype</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Test Equivalence (register-shifted register) performs a bitwise exclusive OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

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

cond

A1

TEQ{<c>}<q>} <Rn>, <Rm>, <type> <Rs>

n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<
c> See Standard assembler syntax fields.
<
q> See Standard assembler syntax fields.
<
Rn> Is the first general-purpose source register, encoded in the "Rn" field.
<
Rm> Is the second general-purpose source register, encoded in the "Rm" field.
<
type> Is the type of shift to be applied to the second source register, encoded in “stype”:

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

<
Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If **FEAT_TRF** is not implemented, this instruction executes as a NOP.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1** (Armv8.4)

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

cond

**A1**

```
TSB{<c>}{<q>} CSYNC
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
if cond != '1110' then UNPREDICTABLE; // ESB must be encoded with AL condition
```

**CONSTRAINED UNPREDICTABLE behavior**

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

**T1** (Armv8.4)

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**T1**

```
TSB{<c>}{<q>} CSYNC
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
if InITBlock() then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*. 
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    TraceSynchronizationBarrier();
TST (immediate)

Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | Rn | [0][0][0][0] | imm12 |
| cond |

A1

TST{<c>}{<q>} <Rn>, #<const>

n = Uint(Rn);
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 1 | 0 | i | 0 | 0 | 0 | 0 | 0 | 1 | Rn | 0 | imm3 | 1 | 1 | 1 | 1 | imm8 |

T1

TST{<c>}{<q>} <Rn>, #<const>

n = Uint(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions for the range of values.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  result = R[n] AND imm32;
  PSTATE.N = result<31>;
  PSTATE.Z = IsZeroBit(result);
  PSTATE.C = carry;
  // PSTATE.V unchanged

TST (immediate)
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

<table>
<thead>
<tr>
<th>Rn</th>
<th>imm5</th>
<th>stype</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>![cond]</td>
<td>![1111]</td>
<td>00000</td>
<td>00001</td>
</tr>
</tbody>
</table>

Rotate right with extend (imm5 == 00000 && stype == 11)

TST{<c>}{<q>} <Rn>, <Rm>, RRX

Shift or rotate by value (!!(imm5 == 00000 && stype == 11))

TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

n = UInt(Rn); m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);

T1

<table>
<thead>
<tr>
<th>Rn</th>
<th>imm3</th>
<th>imm2</th>
<th>stype</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>![cond]</td>
<td>01000</td>
<td>00001</td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

T1

TST{<c>}{<q>} <Rn>, <Rm>

n = UInt(Rn); m = UInt(Rm);
(shift_t, shift_n) = (SRTtype_LSL, 0);

T2

<table>
<thead>
<tr>
<th>Rn</th>
<th>imm3</th>
<th>imm2</th>
<th>stype</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>![cond]</td>
<td>01000</td>
<td>00001</td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

TST{<c>}{<q>} <Rn>, <Rm>, RRX

Shift or rotate by value (!!(imm3 == 000 && imm2 == 00 && stype == 11))

TST{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)

TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

n = UInt(Rn); m = UInt(Rm);
(shift t, shift n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c>
See Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<Rn>
For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift>
Is the type of shift to be applied to the second source register, encoded in “stype”:

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

<amount>
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Test (register-shifted register) performs a bitwise AND operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | Rn | (0)| (0)| (0)| (0)| Rs | 0 | stype | 1 | Rm |

cond

A1

TST{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<type> Is the type of shift to be applied to the second source register, encoded in “stype”:

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

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | != 1111 | 0 1 1 0 0 1 0 1 | Rn | Rd | (1) | (1) | (1) | (1) | 0 | 0 | 0 | 1 | Rm |
|---------------------------------------------|------------------|-------------------|--------|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

cond

A1

UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 1 0 1 0 1 0 0 1 | Rn | 1 1 1 1 | Rd | 0 | 1 | 0 | 0 | Rm |
|---------------------------------------------|------------------|-------------------|--------|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

T1

UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

\[
\text{d} = \text{UInt}(\text{Rd})\; \land \; \text{n} = \text{UInt}(\text{Rn})\; \land \; \text{m} = \text{UInt}(\text{Rm})\; \\
\text{if d} = 15 \; \lor \; \text{n} = 15 \; \lor \; \text{m} = 15 \; \text{then UNPREDICTABLE;}\; \text{// Armv8-A removes UNPREDICTABLE for R13}
\]

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

### Operation

if ConditionPassed() then

    EncodingSpecificOperations();
    \[
    \text{sum1} = \text{UInt}(\text{R}[\text{n}])<15:0> + \text{UInt}(\text{R}[\text{m}])<15:0>;
    \text{sum2} = \text{UInt}(\text{R}[\text{n}])<31:16> + \text{UInt}(\text{R}[\text{m}])<31:16>;
    \text{R}[\text{d}]<15:0> = \text{sum1}<15:0>;
    \text{R}[\text{d}]<31:16> = \text{sum2}<15:0>;
    \text{PSTATE.GE<1:0>} = \text{if sum1} \geq 0x10000 \text{ then '11' else '00'};
    \text{PSTATE.GE<3:2>} = \text{if sum2} \geq 0x10000 \text{ then '11' else '00'};
    \]

### Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

The response of this instruction to asynchronous exceptions does not vary based on:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.
**UADD8**

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets `PSTATE.GE` according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

![Encoding Table]

<table>
<thead>
<tr>
<th>Condition</th>
<th>Rn</th>
<th>Rd</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### T1

![Encoding Table]

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

For more information about the constrained UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

**Assembler Symbols**

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0> = sum1<7:0>;
    R[d]<15:8> = sum2<7:0>;
    R[d]<23:16> = sum3<7:0>;
    R[d]<31:24> = sum4<7:0>;
    PSTATE.GE<0> = if sum1 >= 0x100 then '1' else '0';
    PSTATE.GE<1> = if sum2 >= 0x100 then '1' else '0';
    PSTATE.GE<2> = if sum3 >= 0x100 then '1' else '0';
    PSTATE.GE<3> = if sum4 >= 0x100 then '1' else '0';
```


Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets `PSTATE.GE` according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
Rn | Rd | (1)(1)(1)(1) | 0 | 0 | 1 | 1 | Rm

cond
```

```
A1

UASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

**T1**

```
Rn | Rd | (1)(1)(1)(1) | 0 | 0 | 1 | 1 | Rm

cond
```

```
T1

UASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the constrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0> = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if sum >= 0x10000 then '11' else '00';
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination.
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UBFX**

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

![Table: A1 encoding](image)

```plaintext
A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>widthm1</td>
<td>Rd</td>
<td>lsb</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**cond**

### A1

```plaintext
A1

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

\[\begin{align*}
    d &= UInt(Rd); \\
    n &= UInt(Rn); \\
    lsbit &= UInt(lsb); \\
    widthminus1 &= UInt(widthm1); \\
    \text{if } d == 15 || n == 15 \rightarrow \text{UNPREDICTABLE;}
\end{align*}\]

```

### T1

![Table: T1 encoding](image)

```plaintext
T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**cond**

### T1

```plaintext
T1

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

\[\begin{align*}
    d &= UInt(Rd); \\
    n &= UInt(Rn); \\
    lsbit &= UInt(imm3:imm2); \\
    widthminus1 &= UInt(widthm1); \\
    \text{if } d == 15 || n == 15 \rightarrow \text{UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\end{align*}\]

For more information about the constrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the general-purpose source register, encoded in the "Rn" field.
- `<lsb>` For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field. 
  
  For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.

- `<width>` Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        \[R[d] = \text{ZeroExtend}(R[n]<msbit:lsbit>, 32);\]
    else
        UNPREDICTABLE;
```
CONSTRAINED UNPREDICTABLE behavior

If $\text{msbit} > 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Permanently Undefined generates an Undefined Instruction exception.
The encodings for UDF used in this section are defined as permanently UNDEFINED in the Armv8-A architecture.
However:
• With the T32 instruction set, Arm deprecates using the UDF instruction in an IT block.
• In the A32 instruction set, UDF is not conditional.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\ 1 1 1 0 | 0 1 1 1 1 1 1 | imm12 | 1 1 1 1 | imm4
```

```
UDF{<c>}{<q>} {#}<imm>

imm32 = ZeroExtend(imm12:imm4, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\ 1 1 1 0 | 1 1 1 1 1 0 | imm8
```

```
UDF{<c>}{<q>} {#}<imm>

imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T2

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\ 1 1 1 1 0 | 1 1 1 1 1 1 | imm4 | 1 0 1 0 | imm12
```

```
UDF{<c>}.W {#}<imm> // (<imm> can be represented in T1)

UDF{<c>}{<q>} {#}<imm>

imm32 = ZeroExtend(imm4:imm12, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

Assembler Symbols

```
<c> For encoding A1: see Standard assembler syntax fields. <c> must be AL or omitted.
For encoding T1 and T2: see Standard assembler syntax fields. Arm deprecates using any <c> value
other than AL.

<q> See Standard assembler syntax fields.
<imm> For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the
"imm12:imm4" field. The PE ignores the value of this constant.
```
For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant.

For encoding T2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    UNDEFINED;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected. See Divide instructions for more information about this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

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

\[
\text{if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;}
\]

CONSTRAINED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

T1

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

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

\[
\text{if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

CONSTRAINED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- \(<c>\) See Standard assembler syntax fields.
- \(<q>\) See Standard assembler syntax fields.
- \(<\text{Rd}>\) Is the general-purpose destination register, encoded in the "Rd" field.
<Rn> Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.
<Rm> Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[m]) == 0 then
        result = 0;
    else
        result = RoundTowardsZero(Real(UInt(R[n])) / Real(UInt(R[m])));
    R[d] = result<31:0>;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

![Table](image)

```plaintext
A1

UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1

![Table](image)

```plaintext
T1

UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

### Assembler Symbols

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

### Operation

If `ConditionPassed()` then

```plaintext
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0> = sum1<16:1>;
    R[d]<31:16> = sum2<16:1>;
```

### Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UHADD8**

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

cond

### A1

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |   |   |   |   |   |   |   |

### T1

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

if ConditionPassed() then

```
EncodingSpecificOperations();
sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0>   = sum1<8:1>;
R[d]<15:8>  = sum2<8:1>;
R[d]<23:16> = sum3<8:1>;
R[d]<31:24> = sum4<8:1>;
```

UHADD8  
Page 589
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 1 1 0 0 1 1 1                  Rn  Rd</td>
</tr>
</tbody>
</table>

cond

A1

UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 0 1 0 1 0                  Rn  Rd</td>
</tr>
</tbody>
</table>

T1

UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<
c> See Standard assembler syntax fields.

<
q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0> = diff<16:1>;
    R[d]<31:16> = sum<16:1>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | Rn | Rd | (1)(1)(1)(1) | 0 | 1 | 0 | 1 | Rm |
| cond |

A1

UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

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

if \(d = 15 \) || \(n = 15\) || \(m = 15\) then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1

UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

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

if \(d = 15\) || \(n = 15\) || \(m = 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<\text{c}> \quad \text{See Standard assembler syntax fields.}

<\text{q}> \quad \text{See Standard assembler syntax fields.}

<\text{Rd}> \quad \text{Is the general-purpose destination register, encoded in the "Rd" field.}

<\text{Rn}> \quad \text{Is the first general-purpose source register, encoded in the "Rn" field.}

<\text{Rm}> \quad \text{Is the second general-purpose source register, encoded in the "Rm" field.}

Operation

\[
\text{if ConditionPassed() then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{sum} = \text{UInt}(R[n]<15:0>) + \text{UInt}(R[m]<31:16>);
\]

\[
\text{diff} = \text{UInt}(R[n]<31:16>) - \text{UInt}(R[m]<15:0>);
\]

\[
R[d]<15:0> = \text{sum}<16:1>;
\]

\[
R[d]<31:16> = \text{diff}<16:1>;
\]

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

The response of this instruction to asynchronous exceptions does not vary based on:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.
**UHSUB16**

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | Rn | Rd | (1) | (1) | (1) | (1) | 0 | 1 | 1 | 1 | Rm |

**T1**

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

**Assembler Symbols**

- **<c>** See *Standard assembler syntax fields*.
- **<q>** See *Standard assembler syntax fields*.
- **<Rd>** Is the general-purpose destination register, encoded in the "Rd" field.
- **<Rn>** Is the first general-purpose source register, encoded in the "Rn" field.
- **<Rm>** Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations();
  diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
  diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
  R[d]<15:0> = diff1<16:1>;
  R[d]<31:16> = diff2<16:1>;
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>1111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**cond**

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | Rn | 1  | 1  | 1  | Rd | 0  | 1  | 1  | 0  | Rm |

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 =UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0> = diff1<8:1>;
    R[d]<15:8> = diff2<8:1>;
    R[d]<23:16> = diff3<8:1>;
    R[d]<31:24> = diff4<8:1>;
```

For more information about the constrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*. 

UHSUB8
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

A1

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

T1

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<RdLo> Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi> Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

|            | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111    | 0  | 0  | 0  | 0  | 1  | 0  | 1  | S  | RdHi | RdLo | Rm | 1  | 0  | 0  | 1  | Rn |
| cond       |    |    |    |    |    |    |    |    |        |        |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Flag setting (S == 1)**

UMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

**Not flag setting (S == 0)**

UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

\[
dLo = \text{UInt}(RdLo); \quad dHi = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S == '1');
\]

if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;

if dHi == dLo then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### T1

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>RdLo</td>
<td>RdHi</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**T1**

UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

\[
dLo = \text{UInt}(RdLo); \quad dHi = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = \text{FALSE};
\]

if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;

// Armv8-A removes UNPREDICTABLE for R13

if dHi == dLo then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors.*
Assembler Symbols

<q>  
See Standard assembler syntax fields.

<RdLo>  
Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi>  
Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn>  
Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm>  
Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
  R[dHi] = result<63:32>;
  R[dLo] = result<31:0>;
  if setflags then
    PSTATE.N = result<63>;
    PSTATE.Z = IsZeroBit(result<63:0>);
    // PSTATE.C, PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

• The response of this instruction to 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, UMULLS

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result. In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>RdHi</td>
<td>RdLo</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag setting (S == 1)

UMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Not flag setting (S == 0)

UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

dLo = Uint(RdLo); dHi = Uint(RdHi); n = Uint(Rn); m = Uint(Rm); setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;

CONRADINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>RdLo</td>
<td>RdHi</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1

UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

dLo = Uint(RdLo); dHi = Uint(RdHi); n = Uint(Rn); m = Uint(Rm); setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;

CONRADINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONRADINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<RdLo> Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| != 1111 | 0 1 1 0 0 | 1 1 0 | Rn | Rd | (1)(1)(1)(1)(0) 0 0 1 | Rm |

cond

UQADD16\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm>

d = UINT(Rd);  n = UINT(Rn);  m = UINT(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 1 1 0 1 0 1 0 0 1 | Rn | 1 1 1 1 | Rd | 0 1 0 1 | Rm |

T1

UQADD16\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm>

d = UINT(Rd);  n = UINT(Rn);  m = UINT(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\)  \(\text{See Standard assembler syntax fields.}\)

\(<q>\)  \(\text{See Standard assembler syntax fields.}\)

\(<\text{Rd}>\)  Is the general-purpose destination register, encoded in the "Rd" field.

\(<\text{Rn}>\)  Is the first general-purpose source register, encoded in the "Rn" field.

\(<\text{Rm}>\)  Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  sum1 = UINT(R[n]<15:0>) + UINT(R[m]<15:0>);
  sum2 = UINT(R[n]<31:16>) + UINT(R[m]<31:16>);
  R[d]<15:0> = UnsignedSat(sum1, 16);
  R[d]<31:16> = UnsignedSat(sum2, 16);
UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| != 1111 | 0 1 1 0 | 0 1 1 0 | Rn       | Rd       | (1)(1)(1)(1)| 1 0 0 1 | Rm       |

cond

**A1**

UQADD8{<c>}{<q>}{<Rd>,}{<Rn>,}{<Rm>}

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1 1 1 1 1 0 1 0 1 0 0 0 | Rn       | 1 1 1 1 | Rd       | 0 1 0 1 | Rm       |

**T1**

UQADD8{<c>}{<q>}{<Rd>,}{<Rn>,}{<Rm>}

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

if ConditionPassed() then

EncodingSpecificOperations():

sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);

R[d]<7:0> = UnsignedSat(sum1, 8);
R[d]<15:8> = UnsignedSat(sum2, 8);
R[d]<23:16> = UnsignedSat(sum3, 8);
R[d]<31:24> = UnsignedSat(sum4, 8);
UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|----------------|
| != 1111 | 0 1 1 0 0 1 1 0 | Rn | Rd | (1)(1)(1)(1) | 0 0 1 1 | Rm |
```

cond

```
A1

UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|
| 1 1 1 1 1 0 1 0 1 0 1 0 | Rn | 1 1 1 1 | Rd | 0 1 0 1 | Rm |
```

T1

```
T1

UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0> = UnsignedSat(diff, 16);
    R[d]<31:16> = UnsignedSat(sum, 16);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd | (1)|(1)|(1)|(1) | 0 | 1 | 0 | 1 | Rm |

cond

**A1**

UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

```plaintext
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 1 | 0 | 1 | Rm |

**T1**

UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

```plaintext
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    sum = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0> = UnsignedSat(sum, 16);
    R[d]<31:16> = UnsignedSat(diff, 16);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 \( \leq x \leq 2^{16} - 1 \), and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccc}
\hline
! = 1111 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & Rn & Rd & (1)(1)(1)(1) & 0 & 1 & 1 & 1 & Rm \\
\end{array}
\]

\[\text{cond}\]

A1

\[
\text{UQSUB16}\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>
\]

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

\[
\text{if } d == 15 \text{ || } n == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE;}
\]

T1

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & Rd & 0 & 1 & 0 & 1 & Rm \\
\end{array}
\]

T1

\[
\text{UQSUB16}\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>
\]

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

\[
\text{if } d == 15 \text{ || } n == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if **ConditionPassed()** then

\[
\begin{aligned}
\text{EncodingSpecificOperations}(): \\
\text{diff1} = \text{UInt}(R[n]<15:0>) - \text{UInt}(R[m]<15:0>); \\
\text{diff2} = \text{UInt}(R[n]<31:16>) - \text{UInt}(R[m]<31:16>); \\
R[d]<15:0> &= \text{UnsignedSat}(\text{diff1}, 16); \\
R[d]<31:16> &= \text{UnsignedSat}(\text{diff2}, 16);
\end{aligned}
\]
UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | |

\[\text{cond}\]

A1

UQSUB8\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>

d = \text{UInt}(Rd); \; n = \text{UInt}(Rn); \; m = \text{UInt}(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | | | | | | | | | | | | | | | | | | |

T1

UQSUB8\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>

d = \text{UInt}(Rd); \; n = \text{UInt}(Rn); \; m = \text{UInt}(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.
\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.
\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then

EncodingSpecificOperations():

diff1 = \text{UInt}(R[n]<7:0>) - \text{UInt}(R[m]<7:0>);
diff2 = \text{UInt}(R[n]<15:8>) - \text{UInt}(R[m]<15:8>);
diff3 = \text{UInt}(R[n]<23:16>) - \text{UInt}(R[m]<23:16>);
diff4 = \text{UInt}(R[n]<31:24>) - \text{UInt}(R[m]<31:24>);
R[d]<7:0> = \text{UnsignedSat}(\text{diff1}, 8);
R[d]<15:8> = \text{UnsignedSat}(\text{diff2}, 8);
R[d]<23:16> = \text{UnsignedSat}(\text{diff3}, 8);
R[d]<31:24> = \text{UnsignedSat}(\text{diff4}, 8);
**USAD8**

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>0 1 1 1 1 0 0 0</td>
</tr>
</tbody>
</table>

**A1**

```
USAD8{<c>}{<q>}{<Rd>,} <Rn>, <Rm>
```

\[ \begin{align*}
\text{d} &= \text{UInt}(\text{Rd}); \\
\text{n} &= \text{UInt}(\text{Rn}); \\
\text{m} &= \text{UInt}(\text{Rm}); \\
\text{if} \ d == 15 \ | | \ n == 15 \ | | \ m == 15 \text{ then UNPREDICTABLE;}
\end{align*} \]

### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 1 0 1 1 0 | 1 1 1 | Rn | 1 1 1 1 | Rd | 0 0 0 0 | Rm |

**T1**

```
USAD8{<c>}{<q>}{<Rd>,} <Rn>, <Rm>
```

\[ \begin{align*}
\text{d} &= \text{UInt}(\text{Rd}); \\
\text{n} &= \text{UInt}(\text{Rn}); \\
\text{m} &= \text{UInt}(\text{Rm}); \\
\text{if} \ d == 15 \ | | \ n == 15 \ | | \ m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\end{align*} \]

For more information about the ** CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

### Assembler Symbols

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

### Operation

```perl
if ConditionPassed() then
    EncodingSpecificOperations();
    \text{absdiff1} = \text{Abs}(\text{UInt}(\text{R}[n]<7:0>)) - \text{UInt}(\text{R}[m]<7:0>));
    \text{absdiff2} = \text{Abs}(\text{UInt}(\text{R}[n]<15:8>)) - \text{UInt}(\text{R}[m]<15:8>));
    \text{absdiff3} = \text{Abs}(\text{UInt}(\text{R}[n]<23:16>)) - \text{UInt}(\text{R}[m]<23:16>));
    \text{absdiff4} = \text{Abs}(\text{UInt}(\text{R}[n]<31:24>)) - \text{UInt}(\text{R}[m]<31:24>));
    \text{result} = \text{absdiff1} + \text{absdiff2} + \text{absdiff3} + \text{absdiff4};
    \text{R}[d] = \text{result}<31:0>;
```
Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

<table>
<thead>
<tr>
<th>cond</th>
<th>Rd</th>
<th>!1111</th>
<th>0 1 1 1 1 0 0 0</th>
<th>Rm</th>
<th>!1111</th>
<th>0 0 0 1</th>
<th>Rn</th>
</tr>
</thead>
</table>

if Ra == '1111' then SEE “USAD8”;

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

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

**T1**

<table>
<thead>
<tr>
<th>!1111</th>
<th>0 1 1 1 0 1 1 1</th>
<th>Ra</th>
<th>!1111</th>
<th>0 0 0 0</th>
<th>Rm</th>
</tr>
</thead>
</table>

if Ra == '1111' then SEE “USAD8”;

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

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See [Standard assembler syntax fields](https://example.com).
- `<q>` See [Standard assembler syntax fields](https://example.com).
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<Ra>` Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Operation**

if `ConditionPassed()` then

```
EncodingSpecificOperations();
absdiff1 = Abs(UInt(R[n]<7:0>) - UInt(R[m]<7:0>));
absdiff2 = Abs(UInt(R[n]<15:8>) - UInt(R[m]<15:8>));
absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
R[d] = result<31:0>;
```
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range. This instruction sets PSTATE.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

Arithmetic shift right (sh == 1)

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Logical shift left (sh == 0)

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

if d == 15 || n == 15 then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

Arithmetic shift right (sh == 1 && !(imm3 == 000 && imm2 == 00))

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Logical shift left (sh == 0)

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

if sh == '1' && (imm3:imm2) == '00000' then SEE "USAT16";

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<

See Standard assembler syntax fields.

<

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the "Rd" field.

<imm>

Is the bit position for saturation, in the range 0 to 31, encoded in the "sat_imm" field.

<Rn>

Is the general-purpose source register, encoded in the "Rn" field.

<amount>

For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32.
For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.
For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.
This instruction sets \textit{PSTATE}.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (\textit{A1}) and T32 (\textit{T1}).

\textbf{A1}

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
! = 1111 | 0 1 1 0 1 1 1 0 | sat_imm | Rd | (1)(1)(1)(1) 0 0 1 1 | Rn
\end{verbatim}

\textbf{cond}

\textbf{A1}

\begin{verbatim}
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

\hspace{1cm} d = UInt(Rd); \ n = UInt(Rn); \ saturate\_to = UInt(sat\_imm);
\hspace{1cm} if d == 15 || n == 15 then UNPREDICTABLE;
\end{verbatim}

\textbf{T1}

\begin{verbatim}
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 (0) 1 1 1 0 1 0 | Rn | 0 0 0 0 | Rd | 0 0 (0)(0) sat\_imm
\end{verbatim}

\textbf{T1}

\begin{verbatim}
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

\hspace{1cm} d = UInt(Rd); \ n = UInt(Rn); \ saturate\_to = UInt(sat\_imm);
\hspace{1cm} if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
\end{verbatim}

For more information about the \textit{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see \textit{Architectural Constraints on UNPREDICTABLE behaviors}.

\textbf{Assembler Symbols}

\begin{itemize}
\item <c> See \textit{Standard assembler syntax fields}.
\item <q> See \textit{Standard assembler syntax fields}.
\item <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
\item <imm> Is the bit position for saturation, in the range 0 to 15, encoded in the "sat\_imm" field.
\item <Rn> Is the general-purpose source register, encoded in the "Rn" field.
\end{itemize}

\textbf{Operation}

\begin{verbatim}
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate\_to);
    (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate\_to);
    R[d]<15:0> = ZeroExtend(result1, 16);
    R[d]<31:16> = ZeroExtend(result2, 16);
    if sat1 || sat2 then
        PSTATE.Q = '1';
\end{verbatim}
USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets PSTATE.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| != 1111 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | Rn | Rd | (1) | (1) | (1) | (1) | 0 | 1 | 0 | 1 | Rm |

cond

A1

USAX{<c>}<{q}> {<Rd>}, {<Rn>, <Rm>}

\[d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);\]
\[
\text{if } d == 15 \quad || \quad n == 15 \quad || \quad m == 15 \text{ then UNPREDICTABLE};
\]

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 1 | 0 | 0 | Rm |

T1

USAX{<c>}<{q}> {<Rd>}, {<Rn>, <Rm>}

\[d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);\]
\[
\text{if } d == 15 \quad || \quad n == 15 \quad || \quad m == 15 \text{ then UNPREDICTABLE}; \quad // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then

    EncodingSpecificOperations();
    \[\text{sum} = \text{UInt}(R[n]<15:0>) + \text{UInt}(R[m]<31:16>);\]
    \[\text{diff} = \text{UInt}(R[n]<31:16>) - \text{UInt}(R[m]<15:0>);\]
    \[R[d]<15:0> = \text{sum}<15:0>;\]
    \[R[d]<31:16> = \text{diff}<15:0>;\]
    
PSTATE.GE<1:0> = if \text{sum} == 0x10000 then '11' else '00';
    
PSTATE.GE<3:2> = if \text{diff} == 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

cond

A1

USUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 0 1 1 0 1</td>
</tr>
</tbody>
</table>

T1

USUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<cond> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.
<Rd> Is the general-purpose destination register, encoded in the "Rd" field.
<Rn> Is the first general-purpose source register, encoded in the "Rn" field.
<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0> = diff1<15:0>;
    R[d]<31:16> = diff2<15:0>;
    PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | Rn | Rd | (1) | (1) | (1) | (1) | 1 | 1 | 1 | 1 | Rm |

**cond**

A1

USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td>0</td>
</tr>
</tbody>
</table>

T1

USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the constrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>`  See *Standard assembler syntax fields*.
- `<q>`  See *Standard assembler syntax fields*.
- `<Rd>`  Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>`  Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>`  Is the second general-purpose source register, encoded in the "Rm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0> = diff1<7:0>;
    R[d]<15:8> = diff2<7:0>;
    R[d]<23:16> = diff3<7:0>;
    R[d]<31:24> = diff4<7:0>;
    PSTATE.GE<0> = if diff1 >= 0 then '1' else '0';
    PSTATE.GE<1> = if diff2 >= 0 then '1' else '0';
    PSTATE.GE<2> = if diff3 >= 0 then '1' else '0';
    PSTATE.GE<3> = if diff4 >= 0 then '1' else '0';
```
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| != 1111                                        | != 1111                                        |
| Rd                                            | rotate(0)(0)                                  |
| rotate(0)(0)                                 | 0 1 1 1                                     |
| 0                                             | 1                                              |
| Rm                                            |                                                |
| cond                                          |                                                |
| Rn                                            |                                                |

A1

UXTAB{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

if Rn == '1111' then SEE "UXTB";

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|----------------------------------------|
| != 1111                                | != 1111                                |
| Rd 1(0)rotate                          | Rm                                     |
| Rn                                      |                                        |

T1

UXTAB{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

if Rn == '1111' then SEE "UXTB";

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<

See Standard assembler syntax fields.

<

See Standard assembler syntax fields.

<Rd>

Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>

Is the first general-purpose source register, encoded in the "Rn" field.

<Rm>

Is the second general-purpose source register, encoded in the "Rm" field.

<amount>

Is the rotate amount, encoded in "rotate":

<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UXTAB16

Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

cond Rn

A1

UXTAB16{<c>{<q>} {<Rd>,} <Rn}, <Rm} {, ROR #<amount>}

if Rn == '1111' then SEE “UXTB16”;

if Rn == '1111' then SEE “UXTB16”;
d = Uint(Rd); n = Uint(Rn); m = Uint(Rm); rotation = Uint(rotate:'000');

if d == 15 || m == 15 then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 0</td>
</tr>
</tbody>
</table>

Rn

T1

UXTAB16{<c>{<q>} {<Rd>,} <Rn}, <Rm} {, ROR #<amount>}

if Rn == '1111' then SEE “UXTB16”;

d = Uint(Rd); n = Uint(Rn); m = Uint(Rm); rotation = Uint(rotate:'000');

if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in "rotate":

<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

UXTAB16
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0> = R[n]<15:0> + ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

**cond**

**Rn**

**Rm**

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 0 0 0 1</td>
</tr>
</tbody>
</table>

**Rn**

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 0 0 0 1</td>
</tr>
</tbody>
</table>

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<amount>` Is the rotate amount, encoded in "rotate":

<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>
Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| != 1111 | 0 1 1 0 1 | 1 1 0 | 1 1 1 | Rd | rotate | 0 | 0 | 1 | 1 | 1 | Rm |
```

### T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 1 0 1 1 0 0 1 0 | 1 | 1 | Rm | Rd |
```

### T2

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 1 1 0 1 0 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | Rm | 1 | rotate | Rm |
```

For more information about the constrained unpredictable behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

### Assembler Symbols

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the general-purpose source register, encoded in the "Rm" field.
- `<amount>` Is the rotate amount, encoded in “rotate”: 0, 8, 16, or 24.
<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| ! | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | Rd | rotate[0] | (0) | 0 | 1 | 1 | 1 | Rm |
| cond |

**T1**

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | Rd | 1 | [0] | rotate | Rm |

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the “Rd” field.
- `<Rm>` For encoding A1: is the general-purpose source register, encoded in the “Rm” field. For encoding T1: is the second general-purpose source register, encoded in the “Rm” field.
- `<amount>` Is the rotate amount, encoded in “rotate”:

<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

**Operation**

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0> = ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = ZeroExtend(rotated<23:16>, 16);
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

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

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | Rd | rotate(0)(0) | 0 | 1 | 1 | 1 | Rm |

cond

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | Rm | Rd |

**T2**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | Rd | 1 | 0 | rotate | Rm |

**T2**

UXTH{c}{<q>} {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)

UXTH{c}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

For more information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the general-purpose source register, encoded in the "Rm" field.
- `<amount>` Is the rotate amount, encoded in “rotate”: 

<table>
<thead>
<tr>
<th>rotate</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**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 and Send Event*. As described in *Wait For Event 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 Undefined mode of PL0 execution of WFE and WFI instructions.**
- **Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions.**
- **Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode.**

It has encodings from the following instruction sets: A32 (A1 and T32 (T1 and T2).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

**WFE{<c>}{<q>}**

// No additional decoding required

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 1 1 1 0 0 1 0 0 0 0</td>
</tr>
</tbody>
</table>

**WFE{<c>}{<q>}**

// No additional decoding required

### T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0</td>
</tr>
</tbody>
</table>

**WFE{<c>}.W**

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors.*

**Assembler Symbols**

- **<c>** See *Standard assembler syntax fields.*
- **<q>** See *Standard assembler syntax fields.*
if `ConditionPassed()` then
    EncodingSpecificOperations();
    if `IsEventRegisterSet()` then
        ClearEventRegister();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch32.CheckForWFxTrap(EL1, TRUE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32.CheckForWFxTrap(EL2, TRUE);
        if HaveEL(EL3) && PSTATE.M != M32 Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32.CheckForWFxTrap(EL3, TRUE);
        integer localtimeout = -1;  // No local timeout event is generated
        WaitForEvent(localtimeout);
**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 Undefined mode of PL0 execution of WFE and WFI instructions.**
- **Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions.**
- **Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode.**

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

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

```
WFI{<c>}{<q>}
```

// No additional decoding required

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
WFI{<c>}{<q>}
```

// No additional decoding required

### T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```
WFI{<c>}.W
```

// No additional decoding required

For more information about the CONstrained UNpredictable behavior of this instruction, see [Architectural Constraints on UNpredictable behaviors](#).

**Assembler Symbols**

- `<c>` See [Standard assembler syntax fields](#).
- `<q>` See [Standard assembler syntax fields](#).
if \texttt{ConditionPassed()} then
  EncodingSpecificOperations();
  if \texttt{!InterruptPending()} then
    if \texttt{PSTATE.EL == EL0} then
      // Check for traps described by the OS which may be EL1 or EL2.
      \texttt{AArch32.CheckForWFxTrap(EL1, FALSE)};
    if \texttt{PSTATE.EL IN \{EL0, EL1\} && EL2Enabled()} && \texttt{!IsHost()} then
      // Check for traps described by the Hypervisor.
      \texttt{AArch32.CheckForWFxTrap(EL2, FALSE)};
    if \texttt{HaveEL(EL3)} && \texttt{PSTATE.M != M32 Monitor} then
      // Check for traps described by the Secure Monitor.
      \texttt{AArch32.CheckForWFxTrap(EL3, FALSE)};
    integer localtimeout = -1; // No local timeout event is generated
    \texttt{WaitForInterrupt(localtimeout)};
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*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 0 0 1 1 0 0 0 0 0 0 0 (1) (1) (1) (0) (0) (0) 0 0 0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>

cond

### A1

YIELD{<c>}{<q>}

// No additional decoding required

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0</td>
</tr>
</tbody>
</table>

### T1

YIELD{<c>}{<q>}

// No additional decoding required

### T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1 1 0 1 0 (1) (1) (1) (1) 1 0 (0) (0) (0) 0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>

### T2

YIELD{<c>}.W

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

YIELD
AArch32 -- SIMD&FP Instructions (alphabetic order)

**AESD**: AES single round decryption.

**AESE**: AES single round encryption.

**AESIMC**: AES inverse mix columns.

**AESMC**: AES mix columns.

**FLDM*X (FLDMDBX, FLDMIAX)**: FLDM*X.

**FSTMDBX, FSTMIDX**: FSTMX.

**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.

**VABA**: Vector Absolute Difference and Accumulate.

**VABAL**: Vector Absolute Difference and Accumulate Long.

**VABD (floating-point)**: Vector Absolute Difference (floating-point).

**VABD (integer)**: Vector Absolute Difference (integer).

**VABDL (integer)**: Vector Absolute Difference Long (integer).

**VABS**: Vector Absolute.

**VACGE**: Vector Absolute Compare Greater Than or Equal.

**VACGT**: Vector Absolute Compare Greater Than.

**VACLE**: Vector Absolute Compare Less Than or Equal: an alias of VACGE.

**VACLT**: Vector Absolute Compare Less Than: an alias of VACGT.

**VADD (floating-point)**: Vector Add (floating-point).

**VADD (integer)**: Vector Add (integer).

**VADDDHN**: Vector Add and Narrow, returning High Half.

**VADDL**: Vector Add Long.

**VADDW**: Vector Add Wide.

**VAND (immediate)**: Vector Bitwise AND (immediate): an alias of VBIC (immediate).

**VAND (register)**: Vector Bitwise AND (register).

**VBIC (immediate)**: Vector Bitwise Bit Clear (immediate).
VBIC (register): Vector Bitwise Bit Clear (register).
VBIF: Vector Bitwise Insert if False.
VBIT: Vector Bitwise Insert if True.
VBSL: Vector Bitwise Select.
VCADD: Vector Complex Add.
VCEO (immediate #0): Vector Compare Equal to Zero.
VCEO (register): Vector Compare Equal.
VCGE (immediate #0): Vector Compare Greater Than or Equal to Zero.
VCGE (register): Vector Compare Greater Than or Equal.
VCGT (immediate #0): Vector Compare Greater Than Zero.
VCGT (register): Vector Compare Greater Than.
VCLE (immediate #0): Vector Compare Less Than or Equal to Zero.
VCLE (register): Vector Compare Less Than or Equal: an alias of VCGE (register).
VCLZ: Vector Count Leading Sign Bits.
VCLT (immediate #0): Vector Compare Less Than Zero.
VCLS: Vector Count Leading Zeros.
VCMLA: Vector Complex Multiply Accumulate.
VCMLA (by element): Vector Complex Multiply Accumulate (by element).
VCMP: Vector Compare.
VCMPE: Vector Compare, raising Invalid Operation on NaN.
VCNT: Vector Count Set Bits.
VCVT (between floating-point and fixed-point, Advanced SIMD): Vector Convert between floating-point and fixed-point.
VCVT (between floating-point and fixed-point, floating-point): Convert between floating-point and fixed-point.
VCVT (between floating-point and integer, Advanced SIMD): Vector Convert between floating-point and integer.
VCVT (floating-point to integer, floating-point): Convert floating-point to integer with Round towards Zero.
VCVT (integer to floating-point, floating-point): Convert integer to floating-point.
VCVTA (Advanced SIMD): Vector Convert floating-point to integer with Round to Nearest with Ties to Away.
VCVTA (floating-point): Convert floating-point to integer with Round to Nearest with Ties to Away.
VCVTB: Convert to or from a half-precision value in the bottom half of a single-precision register.
VCVTB (BFloat16): Converts from a single-precision value to a BFloat16 value in the bottom half of a single-precision register.
VCVTM (Advanced SIMD): Vector Convert floating-point to integer with Round towards -Infinity.
VCVTM (floating-point): Convert floating-point to integer with Round towards -Infinity.

VCVTN (Advanced SIMD): Vector Convert floating-point to integer with Round to Nearest.

VCVTN (floating-point): Convert floating-point to integer with Round to Nearest.

VCVTP (Advanced SIMD): Vector Convert floating-point to integer with Round towards +Infinity.

VCVTP (floating-point): Convert floating-point to integer with Round towards +Infinity.

VCVTR: Convert floating-point to integer.

VCVTT: Convert to or from a half-precision value in the top half of a single-precision register.

VCVTT (BFloat16): Converts from a single-precision value to a BFloat16 value in the top half of a single-precision register.

VDIV: Divide.

VDOT (by element): BFloat16 floating-point indexed dot product (vector, by element).

VDOT (vector): BFloat16 floating-point (BF16) dot product (vector).

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

VDUP (scalar): Duplicate vector element to vector.

VEOR: Vector Bitwise Exclusive OR.

VEXT (byte elements): Vector Extract.

VEXT (multibyte elements): Vector Extract: an alias of VEXT (byte elements).

VFMA: Vector Fused Multiply Accumulate.

VFMA (BFloat16, by scalar): BFloat16 floating-point widening multiply-add long (by scalar).

VFMA (BFloat16, vector): BFloat16 floating-point widening multiply-add long (vector).

VFMA (by scalar): Vector Floating-point Multiply-Add Long to accumulator (by scalar).

VFMA (vector): Vector Floating-point Multiply-Add Long to accumulator (vector).

VFMS: Vector Fused Multiply Subtract.

VFMSL (by scalar): Vector Floating-point Multiply-Subtract Long from accumulator (by scalar).

VFMSL (vector): Vector Floating-point Multiply-Subtract Long from accumulator (vector).

VFNMA: Vector Fused Negate Multiply Accumulate.

VFNMS: Vector Fused Negate Multiply Subtract.

VHADD: Vector Halving Add.

VHSUB: Vector Halving Subtract.

VINS: Vector move Insertion.

VICVT: Javascript Convert to signed fixed-point, rounding toward Zero.

VLD1 (multiple single elements): Load multiple single 1-element structures to one, two, three, or four registers.

VLD1 (single element to all lanes): Load single 1-element structure and replicate to all lanes of one register.

VLD1 (single element to one lane): Load single 1-element structure to one lane of one register.

VLD2 (multiple 2-element structures): Load multiple 2-element structures to two or four registers.

VLD2 (single 2-element structure to all lanes): Load single 2-element structure and replicate to all lanes of two registers.
**VLD2** (single 2-element structure to one lane): Load single 2-element structure to one lane of two registers.

**VLD3** (multiple 3-element structures): Load multiple 3-element structures to three registers.

**VLD3** (single 3-element structure to all lanes): Load single 3-element structure and replicate to all lanes of three registers.

**VLD3** (single 3-element structure to one lane): Load single 3-element structure to one lane of three registers.

**VLD4** (multiple 4-element structures): Load multiple 4-element structures to four registers.

**VLD4** (single 4-element structure to all lanes): Load single 4-element structure and replicate to all lanes of four registers.

**VLD4** (single 4-element structure to one lane): Load single 4-element structure to one lane of four registers.

**VLDM, VLDMD, VLDMA**: Load Multiple SIMD&FP registers.

**VLDR** (immediate): Load SIMD&FP register (immediate).

**VLDR** (literal): Load SIMD&FP register (literal).

**VMAX** (floating-point): Vector Maximum (floating-point).

**VMAX** (integer): Vector Maximum (integer).

**VMAXNM**: Floating-point Maximum Number.

**VMIN** (floating-point): Vector Minimum (floating-point).

**VMIN** (integer): Vector Minimum (integer).

**VMINNM**: Floating-point Minimum Number.

**VMLA** (by scalar): Vector Multiply Accumulate (by scalar).

**VMLA** (floating-point): Vector Multiply Accumulate (floating-point).

**VMLA** (integer): Vector Multiply Accumulate (integer).

**VMLAL** (by scalar): Vector Multiply Accumulate Long (by scalar).

**VMLAL** (integer): Vector Multiply Accumulate Long (integer).

**VMLS** (by scalar): Vector Multiply Subtract (by scalar).

**VMLS** (floating-point): Vector Multiply Subtract (floating-point).

**VMLS** (integer): Vector Multiply Subtract (integer).

**VMLSL** (by scalar): Vector Multiply Subtract Long (by scalar).

**VMLSL** (integer): Vector Multiply Subtract Long (integer).

**VMMLA**: BFloat16 floating-point matrix multiply-accumulate.

**VMOV (between general-purpose register and half-precision)**: Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register.

**VMOV (between general-purpose register and single-precision)**: Copy a general-purpose register to or from a 32-bit SIMD&FP register.

**VMOV (between two general-purpose registers and a doubleword floating-point register)**: Copy two general-purpose registers to or from a SIMD&FP register.

**VMOV (between two general-purpose registers and two single-precision registers)**: Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers.

**VMOV (general-purpose register to scalar)**: Copy a general-purpose register to a vector element.

**VMOV (immediate)**: Copy immediate value to a SIMD&FP register.
VMOV (register): Copy between FP registers.

VMOV (register, SIMD): Copy between SIMD registers: an alias of VORR (register).

VMOV (scalar to general-purpose register): Copy a vector element to a general-purpose register with sign or zero extension.

VMOVL: Vector Move Long.

VMOVN: Vector Move and Narrow.

VMOVX: Vector Move extraction.

VMRS: Move SIMD&FP Special register to general-purpose register.

VMSR: Move general-purpose register to SIMD&FP Special register.

VMUL (by scalar): Vector Multiply (by scalar).

VMUL (floating-point): Vector Multiply (floating-point).

VMUL (integer and polynomial): Vector Multiply (integer and polynomial).

VMULL (by scalar): Vector Multiply Long (by scalar).

VMULL (integer and polynomial): Vector Multiply Long (integer and polynomial).

VMVN (immediate): Vector Bitwise NOT (immediate).

VMVN (register): Vector Bitwise NOT (register).

VNEG: Vector Negate.

VNMLA: Vector Negate Multiply Accumulate.

VNMLS: Vector Negate Multiply Subtract.

VNMUL: Vector Negate Multiply.

VORN (immediate): Vector Bitwise OR NOT (immediate): an alias of VORR (immediate).

VORN (register): Vector bitwise OR NOT (register).

VORR (immediate): Vector Bitwise OR (immediate).

VORR (register): Vector bitwise OR (register).

VPADAL: Vector Pairwise Add and Accumulate Long.

VPADD (floating-point): Vector Pairwise Add (floating-point).

VPADD (integer): Vector Pairwise Add (integer).

VPADDL: Vector Pairwise Add Long.

VPMAX (floating-point): Vector Pairwise Maximum (floating-point).

VPMAX (integer): Vector Pairwise Maximum (integer).

VPMIN (floating-point): Vector Pairwise Minimum (floating-point).

VPMIN (integer): Vector Pairwise Minimum (integer).

VPOP: Pop SIMD&FP registers from Stack: an alias of VLDM, VLDMDB, VLDMIA.

VPUSH: Push SIMD&FP registers to Stack: an alias of VSTM, VSTMDB, VSTMIA.

VQABS: Vector Saturating Absolute.

VQADD: Vector Saturating Add.

VQDMLAL: Vector Saturating Doubling Multiply Accumulate Long.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VQDMLSL</td>
<td>Vector Saturating Doubling Multiply Subtract Long.</td>
</tr>
<tr>
<td>VQDMULH</td>
<td>Vector Saturating Doubling Multiply Returning High Half.</td>
</tr>
<tr>
<td>VQDMULL</td>
<td>Vector Saturating Doubling Multiply Long.</td>
</tr>
<tr>
<td>VQMOVN, VQMOVUN</td>
<td>Vector Saturating Move and Narrow.</td>
</tr>
<tr>
<td>VQNEG</td>
<td>Vector Saturating Negate.</td>
</tr>
<tr>
<td>VORDMLAH</td>
<td>Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half.</td>
</tr>
<tr>
<td>VORDMLSH</td>
<td>Vector Saturating Rounding Doubling Multiply Subtract Returning High Half.</td>
</tr>
<tr>
<td>VORDMULH</td>
<td>Vector Saturating Rounding Doubling Multiply Returning High Half.</td>
</tr>
<tr>
<td>VORSHL</td>
<td>Vector Saturating Rounding Shift Left.</td>
</tr>
<tr>
<td>VORSHRN (zero)</td>
<td>Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.</td>
</tr>
<tr>
<td>VORSHRUN, VORSHRUN</td>
<td>Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.</td>
</tr>
<tr>
<td>VOSHL (register)</td>
<td>Vector Saturating Shift Left (register).</td>
</tr>
<tr>
<td>VOSHIL, VOSHLU (immediate)</td>
<td>Vector Saturating Shift Left (immediate).</td>
</tr>
<tr>
<td>VQSHRN (zero)</td>
<td>Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.</td>
</tr>
<tr>
<td>VQSHRUN, VQSHRUN</td>
<td>Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.</td>
</tr>
<tr>
<td>VOSUB</td>
<td>Vector Saturating Subtract.</td>
</tr>
<tr>
<td>VRADDHN</td>
<td>Vector Rounding Add and Narrow, returning High Half.</td>
</tr>
<tr>
<td>VRECPE</td>
<td>Vector Reciprocal Estimate.</td>
</tr>
<tr>
<td>VRECPS</td>
<td>Vector Reciprocal Step.</td>
</tr>
<tr>
<td>VREV16</td>
<td>Vector Reverse in halfwords.</td>
</tr>
<tr>
<td>VREV32</td>
<td>Vector Reverse in words.</td>
</tr>
<tr>
<td>VREV64</td>
<td>Vector Reverse in doublewords.</td>
</tr>
<tr>
<td>VRHADD</td>
<td>Vector Rounding Halving Add.</td>
</tr>
<tr>
<td>VRINTA (Advanced SIMD):</td>
<td>Vector Round floating-point to integer towards Nearest with Ties to Away.</td>
</tr>
<tr>
<td>VRINTA (floating-point):</td>
<td>Round floating-point to integer to Nearest with Ties to Away.</td>
</tr>
<tr>
<td>VRINTM (Advanced SIMD):</td>
<td>Vector Round floating-point to integer towards -Infinity.</td>
</tr>
<tr>
<td>VRINTM (floating-point):</td>
<td>Round floating-point to integer towards -Infinity.</td>
</tr>
<tr>
<td>VRINTN (Advanced SIMD):</td>
<td>Vector Round floating-point to integer to Nearest.</td>
</tr>
<tr>
<td>VRINTN (floating-point):</td>
<td>Round floating-point to integer to Nearest.</td>
</tr>
<tr>
<td>VRINTP (Advanced SIMD):</td>
<td>Vector Round floating-point to integer towards +Infinity.</td>
</tr>
<tr>
<td>VRINTP (floating-point):</td>
<td>Round floating-point to integer towards +Infinity.</td>
</tr>
<tr>
<td>VRINTR</td>
<td>Round floating-point to integer.</td>
</tr>
<tr>
<td>VRINTX (Advanced SIMD):</td>
<td>Vector round floating-point to integer inexact.</td>
</tr>
<tr>
<td>VRINTX (floating-point):</td>
<td>Round floating-point to integer inexact.</td>
</tr>
</tbody>
</table>
**VRINTZ (Advanced SIMD)**: Vector round floating-point to integer towards Zero.

**VRINTZ (floating-point)**: Round floating-point to integer towards Zero.

**VRSHL**: Vector Rounding Shift Left.

**VRSHR**: Vector Rounding Shift Right.

**VRSHR (zero)**: Vector Rounding Shift Right: an alias of VORR (register).

**VRSHRN**: Vector Rounding Shift Right and Narrow.

**VRSHRN (zero)**: Vector Rounding Shift Right and Narrow: an alias of VMOVN.

**VRSORTE**: Vector Reciprocal Square Root Estimate.

**VRSORTS**: Vector Reciprocal Square Root Step.

**VRSRA**: Vector Rounding Shift Right and Accumulate.

**VRSUBHN**: Vector Rounding Subtract and Narrow, returning High Half.

**VSDOT (by element)**: Dot Product index form with signed integers..

**VSDOT (vector)**: Dot Product vector form with signed integers..

**VSELEQ, VSELGE, VSELGT, VSELVS**: Floating-point conditional select.

**VSHL (immediate)**: Vector Shift Left (immediate).

**VSHL (register)**: Vector Shift Left (register).

**VSHLL**: Vector Shift Left Long.

**VSHR**: Vector Shift Right.

**VSHR (zero)**: Vector Shift Right: an alias of VORR (register).

**VSHRN**: Vector Shift Right Narrow.

**VSHRN (zero)**: Vector Shift Right Narrow: an alias of VMOVN.

**VSLI**: Vector Shift Left and Insert.

**VSMLA**: Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix.

**VSORT**: Square Root.

**VSRA**: Vector Shift Right and Accumulate.

**VSRI**: Vector Shift Right and Insert.

**VST1 (multiple single elements)**: Store multiple single elements from one, two, three, or four registers.

**VST1 (single element from one lane)**: Store single element from one lane of one register.

**VST2 (multiple 2-element structures)**: Store multiple 2-element structures from two or four registers.

**VST2 (single 2-element structure from one lane)**: Store single 2-element structure from one lane of two registers.

**VST3 (multiple 3-element structures)**: Store multiple 3-element structures from three registers.

**VST3 (single 3-element structure from one lane)**: Store single 3-element structure from one lane of three registers.

**VST4 (multiple 4-element structures)**: Store multiple 4-element structures from four registers.

**VST4 (single 4-element structure from one lane)**: Store single 4-element structure from one lane of four registers.

**VSTM, VSTMDB, VSTMIA**: Store multiple SIMD&FP registers.

**VSTR**: Store SIMD&FP register.
VSUB (floating-point): Vector Subtract (floating-point).
VSUB (integer): Vector Subtract (integer).
VSUBHN: Vector Subtract and Narrow, returning High Half.
VSUBL: Vector Subtract Long.
VSUBW: Vector Subtract Wide.
VSUDOT (by element): Dot Product index form with signed and unsigned integers (by element).
VSWP: Vector Swap.
VTBL, VTBX: Vector Table Lookup and Extension.
VTRN: Vector Transpose.
VTST: Vector Test Bits.
VUDOT (by element): Dot Product index form with unsigned integers..
VUDOT (vector): Dot Product vector form with unsigned integers..
VUMMLA: Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix.
VUSDOT (by element): Dot Product index form with unsigned and signed integers (by element).
VUSDOT (vector): Dot Product vector form with mixed-sign integers.
VUSMMLA: Widening 8-bit mixed integer matrix multiply-accumulate into 2x2 matrix.
VUZP: Vector Unzip.
VUZP (alias): Vector Unzip: an alias of VTRN.
VZIP: Vector Zip.
VZIP (alias): Vector Zip: an alias of VTRN.
AESD

AES single round decryption.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

AESD.<dt> <Qd>, <Qm>

if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------|-----------------|-----------------|---|---|
| 1 1 1 1 1 1 1 1 1 1 | D 1 1 | size 0 0 | Vd 0 0 1 1 0 | M 0 | Vm |
```

T1

AESD.<dt> <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<dt> Is the data type, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then

EncodingSpecificOperations(); CheckCryptoEnabled32();
op1 = Q[d>>1]; op2 = Q[m>>1];
Q[d>>1] = AESInvSubBytes(AESInvShiftRows(op1 EOR op2));

Operational information

If CPSR.DIT is 1:

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

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

AESE.<dt> <Qd>, <Qm>

if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = Uint(D:Vd); m = Uint(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 1 1 1 1 1 1 | D | 1 | 1 | size | 0 | 0 | Vd | 0 | 0 | 1 | 1 | 0 | 0 | M | 0 | Vm |

T1

AESE.<dt> <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = Uint(D:Vd); m = Uint(M:Vm);

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<dt> Is the data type, encoded in "size":

<table>
<thead>
<tr>
<th>size &lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>1x</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[m>>1];
    Q[d>>1] = AESSubBytes(AESEShiftRows(op1 EOR op2));

Operational information

If CPSR.DIT is 1:

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

AES inverse mix columns.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 1 1 | 1 | 1 | 1 | D | 1 | 1 | size | 0 | 0 | Vd | 0 | 0 | 1 | 1 | 1 | 1 | M | 0 | Vm |

Assembler Symbols

<d> Is the data type, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckCryptoEnabled32();
  Q[d>>1] = AESInvMixColumns(Q[m>>1]);

Operational information

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

AES mix columns.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

AESMC.<dt> <Qd>, <Qm>

if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|----------------|---------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1 1 1 1 1 1 1 | D 1 1 | size | 0 0 | Vd | 0 0 | 1 1 1 | 0 | M | 0 | Vm |

T1

AESMC.<dt> <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<dt> Is the data type, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = AESMixColumns(Q[m>>1]);

Operational information

If CPSR.DIT is 1:

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

FLDMDBX is the Decrement Before variant of this instruction, and FLDMIAX is the Increment After variant. FLDM*X loads multiple SIMD&FP registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.  
Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.  
Depending on settings in the CPACR, NSACR, and HCPTCR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support. It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

FLDMDBX\(<c>\{<q>\} <Rn>\), \(<\text{dreglist}>\)

Increment After \((P == 0 && U == 1)\)

FLDMIAX\(<c>\{<q>\} <Rn>\{\}, \,<\text{dreglist}>\)

if \(P == '0' \&\& U == '0' \&\& W == '0'\) then SEE “Related encodings”;
if \(P == '1' \&\& W == '0'\) then SEE “VLDR”;
if \(P == U \&\& W == '1'\) then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8: '00', 32);
regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see “FLDM*X”.
if n == 15 \&\& (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' \&\& (d+regs) > 16 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \(\text{regs} == 0\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If \(\text{regs} > 16 || (\text{d}+\text{regs}) > 16\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: See Advanced SIMD and floating-point 64-bit move for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move for the A32 instruction set.

Assembler Symbols

\(<c>\)  See Standard assembler syntax fields.
\(<q>\)  See Standard assembler syntax fields.
\(<\text{Rn}>\)  Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.
\(!\)  Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
\(<\text{dreglist}>\)  Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

Operation

if \(\text{ConditionPassed}()\) then
    EncodingSpecificOperations();  \text{CheckVFPEnabled}(\text{TRUE});
    address = if add then \text{R}[n] else \text{R}[n]-imm32;
    for \(r = 0\) to \text{regs}-1
        if single_regs then
            \text{S}[d+r] = MemA[\text{address},4];  address = address+4;
        else
            word1 = MemA[\text{address},4];  word2 = MemA[\text{address}+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            \text{D}[d+r] = if \text{BigEndian}(\text{AccType_ATOMIC}) then word1:word2 else word2:word1;
        if wback then \text{R}[n] = if add then \text{R}[n]+imm32 else \text{R}[n]-imm32;

FLDM*X (FLDMDBX, FLDMIAX)
FSTMDBX, FSTMIAX

FSTMX stores multiple SIMD&FP registers from the Advanced SIMD and floating-point register file to consecutive locations in using an address from a general-purpose register. Arm deprecates use of FSTMDBX and FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccccc}
\hline
!= & 1 & 1 & 1 & 0 & P & U & D & W & 0 & Rn & Vd & 1 & 0 & 1 & 1 & imm8<7:1> & 1 & \hline
\end{array}
\]

cond

Decrement Before (P == 1 && U == 0 && W == 1)

FSTMDBX\{<c>\}{<q>} <Rn>!,<dreglist>

Increment After (P == 0 && U == 1)

FSTMIAX\{<c>\}{<q>} <Rn>!,<dreglist>

if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If regs > 16 || (d+regs) > 16, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

\[
\begin{array}{cccccccccccccccccc}
& 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & P & U & D & W & 0 & Rn & Vd & 1 & 0 & 1 & 1 & imm8<7:1> & 1 & \hline
\end{array}
\]
Decrement Before \((P == 1 && U == 0 && W == 1)\)

\[
\text{FSTMDBX}\{<c>\}{<q>}, \text{<dreglist>}
\]

Increment After \((P == 0 && U == 1)\)

\[
\text{FSTMIAX}\{<c>\}{<q>}, \text{<dreglist>}
\]

\[
\begin{align*}
\text{if } P &= '0' \&\& U = '0' \&\& W = '0' \text{ then SEE "Related encodings";} \\
\text{if } P &= '1' \&\& W = '0' \text{ then SEE "VSTR";} \\
\text{if } P &= U \&\& W = '1' \text{ then UNDEFINED;}
\end{align*}
\]

// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)

\[
\begin{align*}
\text{single_regs} &= \text{FALSE}; \\
\text{add} &= (U = '1'); \\
\text{wback} &= (W = '1'); \\
\text{d} &= \text{UInt}(D:Vd); \\
\text{n} &= \text{UInt}(Rn); \\
\text{imm32} &= \text{ZeroExtend}(\text{imm8}:'00', 32);
\end{align*}
\]

\[
\begin{align*}
\text{regs} &= \text{UInt}(\text{imm8}) \div 2; \\
&= \text{If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".}
\end{align*}
\]

\[
\begin{align*}
\text{if } n &= 15 \&\& (\text{wback} || \text{CurrentInstrSet}() != \text{InstrSet_A32}) \text{ then UNPREDICTABLE;}
\end{align*}
\]

\[
\begin{align*}
\text{if } \text{regs} &= 0 || \text{regs } > 16 || (d+\text{regs}) > 32 \text{ then UNPREDICTABLE;}
\end{align*}
\]

\[
\begin{align*}
\text{if } \text{imm8<0>} &= '1' \&\& (d+\text{regs}) > 16 \text{ then UNPREDICTABLE;}
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(\text{regs } = 0\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If \(\text{regs } > 16 || (d+\text{regs}) > 16\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: See *Advanced SIMD and floating-point 64-bit move* for the T32 instruction set, or *Advanced SIMD and floating-point 64-bit move* for the A32 instruction set.

**Assembler Symbols**

- **<c>** See *Standard assembler syntax fields*.
- **<q>** See *Standard assembler syntax fields*.
- **<Rn>** Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.
- **!** Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
- **<dreglist>** Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D-Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then B[n] else R[n].imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r];  address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
        if wback then R[n] = if add then R[n]+imm32 else R[n].imm32;

SHA1C

SHA1 hash update (choose).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>SHA1C.32 &lt;Qd&gt;, &lt;Qn&gt;, &lt;Qm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>if !HaveSHA1Ext() then UNDEFINED;</td>
</tr>
<tr>
<td>if Q != '1' then UNDEFINED;</td>
</tr>
<tr>
<td>if Vd&lt;0&gt; == '1'</td>
</tr>
<tr>
<td>d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);</td>
</tr>
</tbody>
</table>

T1

<table>
<thead>
<tr>
<th>SHA1C.32 &lt;Qd&gt;, &lt;Qn&gt;, &lt;Qm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>if InITBlock() then UNPREDICTABLE;</td>
</tr>
<tr>
<td>if !HaveSHA1Ext() then UNDEFINED;</td>
</tr>
<tr>
<td>if Q != '1' then UNDEFINED;</td>
</tr>
<tr>
<td>if Vd&lt;0&gt; == '1'</td>
</tr>
<tr>
<td>d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);</td>
</tr>
</tbody>
</table>

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckCryptoEnabled32();
  X = Q[d>>1];
  Y = Q[n<<1]<31:0>; // Note: 32 bits wide
  W = Q[m<<1];
  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);
  Q[d<<1] = X;
Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1H

SHA1 fixed rotate.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

**A1**

SHA1H.32 <Qd>, <Qm>

if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;

\[
d = \text{UInt}(D:Vd); \quad m = \text{UInt}(M:Vm);
\]

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 1 1 1 1 1 1 | D 1 1 | size | 0 1 | Vd | 0 0 | 1 0 1 1 | M 0 | Vm |

**T1**

SHA1H.32 <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;

\[
d = \text{UInt}(D:Vd); \quad m = \text{UInt}(M:Vm);
\]

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

**Assembler Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Qd&gt;</td>
<td>Is the 128-bit name of the SIMD&amp;FP destination register, encoded in the &quot;D:Vd&quot; field as &lt;Qd&gt;*2.</td>
</tr>
<tr>
<td>&lt;Qm&gt;</td>
<td>Is the 128-bit name of the SIMD&amp;FP source register, encoded in the &quot;M:Vm&quot; field as &lt;Qm&gt;*2.</td>
</tr>
</tbody>
</table>

**Operation**

if ConditionPassed() then
\[
\begin{align*}
\text{EncodingSpecificOperations}(); \quad \text{CheckCryptoEnabled32}(); \\
Q[d>>1] &= \text{ZeroExtend}(\text{ROL}(Q[m>>1]<31:0>, 30), 128);
\end{align*}
\]

**Operational information**

If CPSR.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
SHA1M

SHA1 hash update (majority).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

SHA1M.32 <Qd>, <Qn>, <Qm>

if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd);
n = UInt(N:Vn);
m = UInt(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---|
| 1 1 1 0 1 1 1 0 D 1 0 Vn | Vd | 1 1 0 0 N Q M 0 Vm |

T1

SHA1M.32 <Qd>, <Qn>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd);
n = UInt(N:Vn);
m = UInt(M:Vm);

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then
     EncodingSpecificOperations(); CheckCryptoEnabled32();
     X = Q[d>>1];
     Y = Q[n>>1]<31:0>; // Note: 32 bits wide
     W = Q[m>>1];
     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<0> = ROL(Y<0>, 32);
     Q[d>>1] = X;
Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1P

SHA1 hash update (parity).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{ccccccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & D & 0 & 1 & Vn & Vd & 1 & 1 & 0 & 0 & N & Q & M & 0 & Vm \\
\end{array}
\]

A1

SHA1P.32 <Qd>, <Qn>, <Qm>

if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

\[
\begin{array}{ccccccccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & D & 0 & 1 & Vn & Vd & 1 & 1 & 0 & 0 & N & Q & M & 0 & Vm \\
\end{array}
\]

T1

SHA1P.32 <Qd>, <Qn>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then
   EncodingSpecificOperations(); CheckCryptoEnabled32();
   X = Q[d>>1];
   Y = Q[n>>1]<31:0>; // Note: 32 bits wide
   W = Q[m>>1];
   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);
      Q[d>>1] = X;
Operational information

If CPSR.DIT is 1:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1SU0

SHA1 schedule update 0.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

SHA1SU0.32 <Qd>, <Qn>, <Qm>

if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;

d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1   1   1   1   0   1   1   1   0   D   1   1   | Vn               |
| Vd   1   1   0   0   N   Q   M   0   | Vm               |

T1

SHA1SU0.32 <Qd>, <Qn>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;

d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then

EncodingSpecificOperations(); CheckCryptoEnabled32();

op1 = Q[d>>1]; op2 = Q[n>>1]; op3 = Q[m>>1];

op2 = op2<63:0> : op1<127:64>;

Q[d>>1] = op1 EOR op2 EOR op3;

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
SHA1SU1

SHA1 schedule update 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

**A1**

SHA1SU1.32 <Qd>, <Qm>

if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = Uint(D:Vd); m = Uint(M:Vm);

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| D 1 1 | size 1 | 0 | Vd 0 | 0 | 1 | 1 | 1 | 0 | M 0 | Vm |

**T1**

SHA1SU1.32 <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = Uint(D:Vd); m = Uint(M:Vm);

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

**Assembler Symbols**

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[m>>1];
    T = X EOR LSR(Y, 32);
    W0 = ROL(T<31:0>, 1);
    W1 = ROL(T<63:32>, 1);
    W2 = ROL(T<95:64>, 1);
    W3 = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);

**Operational information**

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
SHA256H

SHA256 hash update part 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

SHA256H.32 <Qd>, <Qn>, <Qm>

if ¬HaveSHA256Ext() then UNDEFINED;
if Q !='1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | D  | 0  | 0  | Vn | 1  | Vd | 1  | 1  | 0  | N  | Q  | M  | 0  | Vm |

T1

SHA256H.32 <Qd>, <Qn>, <Qm>

if ¬InitBlock() then UNPREDICTABLE;
if ¬HaveSHA256Ext() then UNDEFINED;
if Q !='1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then
   EncodingSpecificOperations(); CheckCryptoEnabled32();
   X = Q[d>>1]; Y = Q[n>>1]; W = Q[m>>1]; part1 = TRUE;
   Q[d>>1] = SHA256hash(X, Y, W, part1);

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA256H2

SHA256 hash update part 2.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

**T1**

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|-----------------------------------------------|
| 1 1 1 1 1 1 1 0 D 0 1 Vn              | 1 1 0 0 N Q M 0 Vm                           |
```

**Assembler Symbols**

- `<Qd>` is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- `<Qn>` is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.
- `<Qm>` is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.

**Operation**

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[n>>1]; Y = Q[d>>1]; W = Q[m>>1]; part1 = FALSE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);
```

**Operational information**

If CPSR.DIT is 1:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA256SU0

SHA256 schedule update 0.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

```
SHA256SU0.32 <Qd>, <Qm>

if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-----------------|-----------------|
| 1 1 1 1 1 1 1 1 D 1 1 | size 1 0 | Vd 0 0 1 1 1 1 M 0 | Vm |
```

T1

```
SHA256SU0.32 <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[m>>1];
    T = Y<31:0> : X<127:32>;
    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[X, e, 32];
    Q[d>>1] = result;
```

Operational information

If CPSR.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
SHA256SU1

SHA256 schedule update 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1 SHA256SU1.32 <Qd>, <Qn>, <Qm>

```c
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 0 | D | 1 0 | Vn | Vd | 1 1 0 0 | N | Q | M | 0 | Vm
```

T1 SHA256SU1.32 <Qd>, <Qn>, <Qm>

```c
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd> * 2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn> * 2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm> * 2.
Operation

if ConditionPassed() then
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[n>>1]; Z = Q[m>>1];
    T0 = Z<31:0> : Y<127:32>;
    T1 = Z<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[X, e, 32] + Elem[T0, e, 32];
        Elem[result, e, 32] = elt;
    T1 = result<63:0>;
    for e = 2 to 3
        elt = Elem[T1, e - 2, 32];
        elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
        elt = elt + Elem[X, e, 32] + Elem[T0, e, 32];
        Elem[result, e, 32] = elt;
    Q[d>>1] = result;

Operational information

If CPSR.DIT is 1:

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

Vector Absolute Difference and Accumulate subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector. Operand and result elements are all integers of the same length.

Depending on settings in the **CPACR, NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | VABA{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm>}
---|---|---|---|---|---|
| 1 1 1 1 0 0 1 | U | 0 | D | size | Vn | Vd | 0 1 1 1 | N | Q | M | 1 | Vm |

64-bit SIMD vector (**Q == 0**)

VABA{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm>}

128-bit SIMD vector (**Q == 1**)

VABA{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm>}

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1'); long destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | VABA{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm>}
---|---|---|---|---|---|
| 1 1 1 | U | 1 1 1 1 | 0 | D | size | Vn | Vd | 0 1 1 1 | N | Q | M | 1 | Vm |

64-bit SIMD vector (**Q == 0**)

VABA{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm>}

128-bit SIMD vector (**Q == 1**)

VABA{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm>}

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1'); long destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**Assembler Symbols**

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “U:size”: 64-bit SIMD vector (Q == 0)

VABA{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm>}

128-bit SIMD vector (Q == 1)

VABA{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm>}

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1'); long destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
### Operation

If `ConditionPassed()` then

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r], e, esize];
            op2 = Elem[Din[m+r], e, esize];
            absdiff = Abs(Int(op1, unsigned) - Int(op2, unsigned));
        if long_destination then
            Elem[Q[d>>1], e, 2*esize] = Elem[Qin[d>>1], e, 2*esize] + absdiff;
        else
            Elem[D[d+r], e, esize] = Elem[Din[d+r], e, esize] + absdiff;
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Absolute Difference and Accumulate Long subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 | U | 1 | D != 11 | Vn | Vd | 0 1 0 1 | N | 0 | M | 0 | Vm
```

size

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 U 1 1 1 1 1 | D != 11 | Vn | Vd | 0 1 0 1 | N | 0 | M | 0 | Vm
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

<dt> Is the data type for the elements of the operands, encoded in “U:size”: 

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>00</td>
<td>U8</td>
</tr>
<tr>
<td>01</td>
<td>U16</td>
</tr>
<tr>
<td>10</td>
<td>U32</td>
</tr>
</tbody>
</table>
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();  
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs[Int(op1,unsigned) - Int(op2,unsigned)];
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Absolute Difference (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are floating-point numbers of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 0  | D  | 1  | sz | Vn | Vd | 1  | 1  | 0  | 1  | N  | Q  | M  | 0  | Vm |

64-bit SIMD vector (Q == 0)

VABD{<c}>{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VABD{<c}>{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 0  | D  | 1  | sz | Vn | Vd | 1  | 1  | 0  | 1  | N  | Q  | M  | 0  | Vm |

64-bit SIMD vector (Q == 0)

VABD{<c}>{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VABD{<c}>{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & !HaveFP16Ext() then UNDEFINED;
if sz == '1' & InITBlock() then UNPREDICTABLE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' & InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Assembler Symbols

<
 For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<dt>
Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vm" field.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if `ConditionPassed()` then
  EncodingSpecificOperations(); ReleaseAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
      Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2,StandardFPSCRValue()));

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

• The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**VABD (integer)**

Vector Absolute Difference (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector. 

Operand and result elements are all integers of the same length. Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see [*Enabling Advanced SIMD and floating-point support*](#).

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

**A1**

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

**64-bit SIMD vector (Q == 0)**

VABD<(<c>){<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VABD<(<c>){<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if size == '11' then UNDEFINED; 
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED; 
unsigned = (U == '1'); 
long destination = FALSE; 
esize = 8 << UInt(size); 
elements = 64 DIV esize; 
d = UInt(D:Vd); 
n = UInt(N:Vn); 
m = UInt(M:Vm); 
regs = if Q == '0' then 1 else 2;

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | U | D | size | Vn | Vd | 0 | 1 | 1 | 1 | N | Q | M | 0 | Vm |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 1 | 1 | 0 | D | size | Vn | Vd | 0 | 1 | 1 | 1 | N | Q | M | 0 | Vm |

**64-bit SIMD vector (Q == 0)**

VABD<(<c>){<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VABD<(<c>){<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if size == '11' then UNDEFINED; 
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED; 
unsigned = (U == '1'); 
long destination = FALSE; 
esize = 8 << UInt(size); 
elements = 64 DIV esize; 
d = UInt(D:Vd); 
n = UInt(N:Vn); 
m = UInt(M:Vm); 
regs = if Q == '0' then 1 else 2;

**Assembler Symbols**

<**c**> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. 
For encoding T1: see *Standard assembler syntax fields*.

<**q**> See *Standard assembler syntax fields*.

<**dt**> Is the data type for the elements of the operands, encoded in “U:size”: 

---

*For more information see [*Enabling Advanced SIMD and floating-point support*](#).*
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Absolute Difference Long (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 1 1 0 0 1 1 U 1 D != 11 11 1 Vn | Vd | 1 1 1 1 N 0 M 0 Vm |
| size |

A1

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 1 U 1 1 1 1 1 1 D != 11 11 1 Vn | Vd | 1 1 1 1 N 0 M 0 Vm |
| size |

T1

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operands, encoded in "U:size":

| U size <dt> |
|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 00 | S8 |
| 0 01 | S16 |
| 0 10 | S32 |
| 1 00 | U8 |
| 1 01 | U16 |
| 1 10 | U32 |
<Qd> is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn> is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Absolute takes the absolute value of each element in a vector, and places the results in a second vector. The floating-point version only clears the sign bit. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

64-bit SIMD vector (Q == 0)

VABS{<c>}{<q>]}.dt <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VABS{<c>}{<q>}.dt <Dd>, <Dm>

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(Vd:D); m = UInt(Vm:M); regs = if Q == '0' then 1 else 2;
```

A2

```
!= 1111 | 1 1 1 0 1 | D 1 1 | 0 0 0 0 | Vd | 1 0 | size 1 1 | M | 0 | Vm | cond
```

Half-precision scalar (size == 01)
(Armv8.2)

VABS{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VABS{<c>}{<q>}.F64 <Dd>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```
CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

64-bit SIMD vector (Q == 0)

VABS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VABS{<c>}{<q>}.<dt> <Dd>, <Dm>

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
else = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

Half-precision scalar (size == 01)
(Armv8.2)

VABS{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VABS{<c>}{<q>}.F64 <Dd>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
CONSTRANDED UNPREDICTABLE behavior

If $size == '01'$ && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>
For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding A2, T1 and T2: see Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<dt>
Is the data type for the elements of the vectors, encoded in "F:size":

<table>
<thead>
<tr>
<th></th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Db>
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Sd>
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm>
Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPAbs(Elem[D[m+r],e,esize]);
            else
                result = Abs(SInt(Elem[D[m+r],e,esize]));
                Elem[D[d+r],e,esize] = result<esize-1:0>;
            else
                case esize of
                    when 16 $d = Zeros(16) : FPAbs($m<15:0>);
                    when 32 $d = FPAbs($m); 
                    when 64 $d = FPAbs(D[m]);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Vector Absolute Compare Greater Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros. The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VACLE.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```assembly
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 1 0 D 0 sz Vn Vd 1 1 1 0 N Q M 1 Vm
```

**64-bit SIMD vector (Q == 0)**

```
VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

**128-bit SIMD vector (Q == 1)**

```
VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```assembly
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

**T1**

```assembly
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 0 D 0 sz Vn Vd 1 1 1 0 N Q M 1 Vm
```

**64-bit SIMD vector (Q == 0)**

```
VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

**128-bit SIMD vector (Q == 1)**

```
VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```assembly
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
or_equal = (op == '0');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```
CONSTRAINED UNPREDICTABLE behavior

If \( sz == '1' \) \&\& InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<dt>\) Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

\(<Qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

\(<Qn>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

\(<Qm>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then

\( \text{EncodingSpecificOperations(); CheckAdvSIMDEnabled();} \)
for r = 0 to regs-1

for e = 0 to elements-1

\( \text{op1 = FPAbs(Elem}[D[n+r],e,esize]); op2 = FPAbs(Elem}[D[m+r],e,esize]); \)

if or_equal then

\( \text{test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());} \)

else

\( \text{test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());} \)

\( \text{Elem}[D[d+r],e,esize] = \text{if test_passed then Ones(esize) else Zeros(esize);} \)

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Absolute Compare Greater Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros. The operands and result can be quadword or doubleword vectors. They must all be the same size. The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VACL{T}. It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

64-bit SIMD vector (Q == 0)

VACGT{<c>}{<q>}.{<dt>}{<Qd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VACGT{<c>}{<q>}.{<dt>}{<Qd>, }<Qn>, <Qm>

T1

64-bit SIMD vector (Q == 0)

VACGT{<c>}{<q>}.{<dt>}{<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VACGT{<c>}{<q>}.{<dt>}{<Qd>, }<Qn>, <Qm>
CONSTRANED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = FPAbs(Elem[D[n+r],e,esize]); op2 = FPAbs(Elem[D[m+r],e,esize]);
        if or_equal then
            test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
        else
            test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
        Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  o The values of the data supplied in any of its registers.
  o The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  o The values of the data supplied in any of its registers.
  o The values of the NZCV flags.
Vector Absolute Compare Less Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of \texttt{VACGE}. This means:

- The encodings in this description are named to match the encodings of \texttt{VACGE}.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of \texttt{VACGE} gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (\texttt{A1}) and T32 (\texttt{T1}).

### A1

64-bit SIMD vector (\(Q == 0\))

\[ \text{VACLE}\{<c>\}{<q>}\{<dt>\} \{<Dd>, \}<Dn>, <Dm} \]

is equivalent to

\[ \text{VACGE}\{<c>\}{<q>}\{<dt>\} <Dd>, <Dm}, <Dn} \]

128-bit SIMD vector (\(Q == 1\))

\[ \text{VACLE}\{<c>\}{<q>}\{<dt>\} \{<Qd>, \} <Qn}, <Qm} \]

is equivalent to

\[ \text{VACGE}\{<c>\}{<q>}\{<dt>\} <Qd}, <Qm}, <Qn} \]

### T1

64-bit SIMD vector (\(Q == 0\))

\[ \text{VACLE}\{<c>\}{<q>}\{<dt>\} \{<Dd>, \}<Dn}, <Dm} \]

is equivalent to

\[ \text{VACGE}\{<c>\}{<q>}\{<dt>\} <Dd}, <Dm}, <Dn} \]

128-bit SIMD vector (\(Q == 1\))

\[ \text{VACLE}\{<c>\}{<q>}\{<dt>\} \{<Qd>, \} <Qn}, <Qm} \]

is equivalent to

\[ \text{VACGE}\{<c>\}{<q>}\{<dt>\} <Qd}, <Qm}, <Qn} \]

**Assembler Symbols**

- \(<Dm>\) is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \( <Qm> \times 2 \).

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \( <Qn> \times 2 \).

For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

See *Standard assembler syntax fields*.

Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \( <Qd> \times 2 \).

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**Operation**

The description of *VACGE* gives the operational pseudocode for this instruction.
**VACLT**

Vector Absolute Compare Less Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of **VACGT**. This means:

- The encodings in this description are named to match the encodings of **VACGT**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **VACGT** gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>D</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Vn</th>
<th>Vd</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>Q</th>
<th>M</th>
<th>1</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**64-bit SIMD vector** (Q == 0)

VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

**128-bit SIMD vector** (Q == 1)

VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

### T1

| D | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | Vn | Vd | 1 | 1 | 1 | 0 | N | Q | M | 1 | Vm |
|---|---|---|---|---|---|---|---|---|---|----|----|---|---|---|---|---|---|---|---|
| op |

**64-bit SIMD vector** (Q == 0)

VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

**128-bit SIMD vector** (Q == 1)

VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

**Assembler Symbols**

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>\times 2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>\times 2.

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

See Standard assembler syntax fields.

Is the data type for the elements of the vectors, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\times 2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**Operation**

The description of VACGT gives the operational pseudocode for this instruction.
VADD (floating-point)

Vector Add (floating-point) adds corresponding elements in two vectors, and places the results in the destination vector. Depending on settings in the **CPACR**, **NSACR**, **HCPTR**, and **FPEXC** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see **Enabling Advanced SIMD and floating-point support**.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

### A1

<table>
<thead>
<tr>
<th>sz</th>
<th>Vn</th>
<th>Vd</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>D</th>
<th>0</th>
<th>sz</th>
<th>Vn</th>
<th>Vd</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>N</th>
<th>Q</th>
<th>M</th>
<th>0</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**64-bit SIMD vector (Q == 0)**

VADD{<c>}{<q>}.<dt> {<Dd>,} {<Dn>,} {<Dm>}

**128-bit SIMD vector (Q == 1)**

VADD{<c>}{<q>}.<dt> {<Qd>,} {<Qn>,} {<Qm>}

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

### A2

<table>
<thead>
<tr>
<th>sz</th>
<th>Vn</th>
<th>Vd</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>D</th>
<th>1</th>
<th>1</th>
<th>Vn</th>
<th>Vd</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>N</th>
<th>0</th>
<th>M</th>
<th>0</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>N</td>
<td>0</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
</tr>
</tbody>
</table>

**Half-precision scalar (size == 01)**

(Armv8.2)

VADD{<c>}{<q>}.F16 {<Sd>,} {<Sn>,} {<Sm>}

**Single-precision scalar (size == 10)**

VADD{<c>}{<q>}.F32 {<Sd>,} {<Sn>,} {<Sm>}

**Double-precision scalar (size == 11)**

VADD{<c>}{<q>}.F64 {<Dd>,} {<Dn>,} {<Dm>}

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D);  n = UInt(Vn:N);  m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D);  n = UInt(Vn:N);  m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | D  | 0  | sz | Vn | Vd | 1  | 1  | 0  | 1  | N  | Q  | M  | 0  | Vm |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

64-bit SIMD vector (Q == 0)

VADD{<c>}{<q>}.<dt>{<Qd>}, }<Qn>, <Qm>

128-bit SIMD vector (Q == 1)

VADD{<c>}{<q>}.<dt>{<Qd>}, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>D</th>
<th>1</th>
<th>1</th>
<th>Vn</th>
<th>Vd</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>N</th>
<th>0</th>
<th>M</th>
<th>0</th>
<th>Vm</th>
</tr>
</thead>
</table>

VADD (floating-point) Page 705
**Half-precision scalar (size == 01)**

VADD\{<c>\}{<q>}.F16 \{<Sd>, <Sn>, <Sm>\}

**Single-precision scalar (size == 10)**

VADD\{<c>\}{<q>}.F32 \{<Sn>, <Sm>\}

**Double-precision scalar (size == 11)**

VADD\{<c>\}{<q>}.F64 \{<Dd>, <Dn>, <Dm>\}

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler Symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  For encoding A2, T1 and T2: see *Standard assembler syntax fields*.

- `<q>` See *Standard assembler syntax fields*.

- `<dt>` Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- `<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- `<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVPFEnabled(TRUE, advsimd);
if advsimd then  // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPAdd(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
else             // VFP instruction
    case esize of
        when 16
            S[d] = Zeros(16) : FPAdd(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        when 32
            S[d] = FPAdd(S[n], S[m], FPSCR[]);
        when 64
            D[d] = FPAdd(D[n], D[m], FPSCR[]);

**VADD (integer)**

Vector Add (integer) adds corresponding elements in two vectors, and places the results in the destination vector. Depending on settings in the `CPACR`, `NSACR`, and `HCPTR` registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see [*Enabling Advanced SIMD and floating-point support*](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

**64-bit SIMD vector (Q == 0)**

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**T1**

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector (Q == 0)**

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**Assembler Symbols**

- `<c>`: For encoding A1: see [*Standard assembler syntax fields*](#). This encoding must be unconditional.
- `<q>`: For encoding T1: see [*Standard assembler syntax fields*](#).
- `<dt>`: Is the data type for the elements of the vectors, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>16</td>
</tr>
<tr>
<td>01</td>
<td>116</td>
</tr>
<tr>
<td>10</td>
<td>132</td>
</tr>
<tr>
<td>11</td>
<td>164</td>
</tr>
</tbody>
</table>

- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.`
- `<Qn>`: Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.`
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

---

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  # CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] + Elem[D[m+r],e,esize];
```

---

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are truncated. For rounded results, see VRADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTOR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

VADDHN{<c>}{<q>}<dt> <Qn>, <Qm>

if size == '11' then SEE “Related encodings”;
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << Uint(size);  elements = 64 DIV esize;
d = Uint(D:Vd);  n = Uint(N:Vn);  m = Uint(M:Vm);

T1

VADDHN{<c>}{<q>}<dt> <Qn>, <Qm>

if size == '11' then SEE “Related encodings”;
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << Uint(size);  elements = 64 DIV esize;
d = Uint(D:Vd);  n = Uint(N:Vn);  m = Uint(M:Vm);

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operands, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>116</td>
</tr>
<tr>
<td>01</td>
<td>132</td>
</tr>
<tr>
<td>10</td>
<td>164</td>
</tr>
</tbody>
</table>

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1
    result = \text{Elem}[\text{Qin}[n>>1],e,2*esize] + \text{Elem}[\text{Qin}[m>>1],e,2*esize];
    \text{Elem}[D[d],e,esize] = result<2*esize-1:esize>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Add Long adds corresponding elements in two doubleword vectors, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of both operands. Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see **Enabling Advanced SIMD and floating-point support**.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 1 1 1 1 0 0 1 U 1 D != 11 | Vn | Vd | 0 0 0 0 N 0 M 0 | Vm |

size op

if size == '11' then SEE “Related encodings”;
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 1 1 1 U 1 1 1 1 1 D != 11 | Vn | Vd | 0 0 0 0 N 0 M 0 | Vm |

size op

if size == '11' then SEE “Related encodings”;
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

Related encodings: See **Advanced SIMD data-processing** for the T32 instruction set, or **Advanced SIMD data-processing** for the A32 instruction set.

Assembler Symbols

For encoding A1: see **Standard assembler syntax fields**. This encoding must be unconditional.
For encoding T1: see **Standard assembler syntax fields**.

See **Standard assembler syntax fields**.

Is the data type for the elements of the second operand vector, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Qin][n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din][n],e,esize], unsigned);
        result = op1 + Int(Elem[Din][m],e,esize],unsigned);
        Elem[Q][d>>1],e,2*esize] = result<2*esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VADDW

Vector Add Wide adds corresponding elements in one quadword and one doubleword vector, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of the doubleword operand. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 | U | 1 | D | != 11 | Vn | Vd | 0 0 0 1 | N | 0 | M | 0 | Vm

size op

A1

VADDW{<c>}{<q>}.{<dt>} {<Qd>}, {<Qn>}, {<Dm>}

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 | U | 1 1 1 1 1 | D | != 11 | Vn | Vd | 0 0 0 1 | N | 0 | M | 0 | Vm

size op

T1

VADDW{<c>}{<q>}.{<dt>} {<Qd>}, {<Qn>}, {<Dm>}

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the second operand vector, encoded in "U:size":

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Qin]\[n>>1\],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din][n],e,esize], unsigned);
    result = op1 + Int(Elem[Din][m],e,esize],unsigned);
    Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Bitwise AND (register) performs a bitwise AND operation between two registers, and places the result in the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VAND{<c>}{<q>}{<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VAND{<c>}{<q>}{<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
    d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 1 0 1 1 1 1 0 D 0 0 | Vn | Vd | 0 0 0 1 | N | Q | M | 1 | Vm |

64-bit SIMD vector (Q == 0)

VAND{<c>}{<q>}{<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VAND{<c>}{<q>}{<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
    d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND D[m+r];
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Bitwise AND (immediate) performs a bitwise AND between a register value and an immediate value, and returns the result into the destination vector.

This is a pseudo-instruction of VBIC (immediate). This means:

- The encodings in this description are named to match the encodings of VBIC (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VBIC (immediate) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

64-bit SIMD vector (Q == 0)

\[
\text{VAND\{<c>\}{<q>}.I16\{<Dd>,\} <Dd>, \#<imm>}
\]

is equivalent to

\[
\text{VBIC\{<c>\}{<q>}.I16 <Dd>, \#<imm>}
\]

128-bit SIMD vector (Q == 1)

\[
\text{VAND\{<c>\}{<q>}.I16\{<Qd>,\} <Qd>, \#<imm>}
\]

is equivalent to

\[
\text{VBIC\{<c>\}{<q>}.I16 <Qd>, \#<imm>}
\]

A2

64-bit SIMD vector (Q == 0)

\[
\text{VAND\{<c>\}{<q>}.I32\{<Dd>,\} <Dd>, \#<imm>}
\]

is equivalent to

\[
\text{VBIC\{<c>\}{<q>}.I32 <Dd>, \#<imm>}
\]

128-bit SIMD vector (Q == 1)

\[
\text{VAND\{<c>\}{<q>}.I32\{<Qd>,\} <Qd>, \#<imm>}
\]

is equivalent to

\[
\text{VBIC\{<c>\}{<q>}.I32 <Qd>, \#<imm>}
\]

T1
64-bit SIMD vector (Q == 0)

$$\text{VAND}\{<c>\}{<q>}.I16 \{<Dd>,\} <Dd>, ~<\text{imm}>$$

is equivalent to

$$\text{VBIC}\{<c>\}{<q>}.I16 <Dd>, ~<\text{imm}>$$

128-bit SIMD vector (Q == 1)

$$\text{VAND}\{<c>\}{<q>}.I16 \{<Qd>,\} <Qd>, ~<\text{imm}>$$

is equivalent to

$$\text{VBIC}\{<c>\}{<q>}.I16 <Qd>, ~<\text{imm}>$$

T2

64-bit SIMD vector (Q == 0)

$$\text{VAND}\{<c>\}{<q>}.I32 \{<Dd>,\} <Dd>, ~<\text{imm}>$$

is equivalent to

$$\text{VBIC}\{<c>\}{<q>}.I32 <Dd>, ~<\text{imm}>$$

128-bit SIMD vector (Q == 1)

$$\text{VAND}\{<c>\}{<q>}.I32 \{<Qd>,\} <Qd>, ~<\text{imm}>$$

is equivalent to

$$\text{VBIC}\{<c>\}{<q>}.I32 <Qd>, ~<\text{imm}>$$

Assembler Symbols

$c$ For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1 and T2: see Standard assembler syntax fields.

$q$ See Standard assembler syntax fields.

$Qd$ Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as $Qd*$2.

$Dd$ Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

$imm$ Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of $imm$, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

Operation

The description of VBIC (immediate) gives the operational pseudocode for this instruction.
VBIC (immediate)

Vector Bitwise Bit Clear (immediate) performs a bitwise AND between a register value and the complement of an immediate value, and returns the result into the destination vector. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VAND (immediate).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

| 31 30 29 28 27 26 25 24 23 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 | i | 1 | D | 0 | 0 | 0 | imm3 | Vd | 0 | x | x | 1 | 0 | Q | 1 | 1 | imm4 |
| cmode |

64-bit SIMD vector (Q == 0)

VBIC{<c><q>}.I32 {<Dd>,} <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VBIC{<c><q>}.I32 {<Qd>,} <Qd>, #<imm>

if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' & & Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEncodeImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

A2

| 31 30 29 28 27 26 25 24 23 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 | i | 1 | D | 0 | 0 | 0 | imm3 | Vd | 1 | 0 | x | 1 | 0 | Q | 1 | 1 | imm4 |
| cmode |

64-bit SIMD vector (Q == 0)

VBIC{<c><q>}.I16 {<Dd>,} <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VBIC{<c><q>}.I16 {<Qd>,} <Qd>, #<imm>

if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' & & Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEncodeImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| i | 1 | 1 | 1 | 1 | 1 | 1 | D | 0 | 0 | 0 | imm3 | Vd | 0 | x | x | 1 | 0 | Q | 1 | 1 | imm4 |
| cmode |
64-bit SIMD vector (Q == 0)

VBIC{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VBIC{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>

if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 1 1 1 1 1 1 1 1 | D | 0 | 0 | 0 | imm3 | Vd | 1 | 0 | x | 1 | 0 | Q | 1 | 1 | imm4 |

cmode

64-bit SIMD vector (Q == 0)

VBIC{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VBIC{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>

if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>  For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<imm>  Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] AND NOT(imm64);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:
• The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Vector Bitwise Bit Clear (register) performs a bitwise AND between a register value and the complement of a register value, and places the result in the destination register. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>64-bit SIMD vector (Q == 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1</td>
<td>D</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 1)

VBIC{<c>}{<q>}{<dt>} {<Dd>,} {Dn,} <Dm>

128-bit SIMD vector (Q == 1)

VBIC{<c>}{<q>}{<dt>} {<Qd>,} {Qn,} <Qm>

```plaintext
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
```

```plaintext
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>64-bit SIMD vector (Q == 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 1)

VBIC{<c>}{<q>}{<dt>} {<Dd>,} {Dn,} <Dm>

128-bit SIMD vector (Q == 1)

VBIC{<c>}{<q>}{<dt>} {<Qd>,} {Qn,} <Qm>

```plaintext
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
```

```plaintext
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

### Assembler Symbols

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  - For encoding T1: see *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.
- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- `<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.
- `<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
Operation

\[
\text{if } \text{ConditionPassed}() \text{ then}
\]
\[
\text{EncodingSpecificOperations(); CheckAdvSIMDEnabled();}
\]
\[
\text{for } r = 0 \text{ to } \text{regs-1}
\]
\[
D[d+r] = D[n+r] \text{ AND NOT}(D[m+r]);
\]

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

op

### 64-bit SIMD vector (Q == 0)

VBIF{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

### 128-bit SIMD vector (Q == 1)

VBIF{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | D  | 1  | 1  | Vn | Vd | 0  | 0  | 0  | 1  | N  | Q  | M  | 1  | Vm |

op

### 64-bit SIMD vector (Q == 0)

VBIF{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

### 128-bit SIMD vector (Q == 1)

VBIF{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

### Assembler Symbols

- `<c>` For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
- For encoding T1: see Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```c
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
        when VBitOps_VBIF D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
        when VBitOps_VBIT D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
        when VBitOps_VBSL D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Bitwise Insert if True inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 1, otherwise leaves the bit in the destination register unchanged. Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

**64-bit SIMD vector (Q == 0)**

VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 1 1 1 1 1 0 D 1 0 | Vn | Vd | 0 0 0 1 | N | Q | M | 1 | Vm |

**64-bit SIMD vector (Q == 0)**

VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**Assembler Symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  For encoding T1: see *Standard assembler syntax fields*.

- `<q>` See *Standard assembler syntax fields*.

- `<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT   D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL   D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Bitwise Select sets each bit in the destination to the corresponding bit from the first source operand when the original destination bit was 1, otherwise from the second source operand. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

**64-bit SIMD vector (Q == 0)**

VBSL{<c>}{<q>}{<dt>} {<Dd>,} {<Dn>, <Dm>}

**128-bit SIMD vector (Q == 1)**

VBSL{<c>}{<q>}{<dt>} {<Qd>,} {<Qn>, <Qm>}

```plaintext
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>Vn</td>
<td>Vd</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector (Q == 0)**

VBSL{<c>}{<q>}{<dt>} {<Dd>,} {<Dn>, <Dm>}

**128-bit SIMD vector (Q == 1)**

VBSL{<c>}{<q>}{<dt>} {<Qd>,} {<Qn>, <Qm>}

```plaintext
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

**Assembler Symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  
  For encoding T1: see *Standard assembler syntax fields*.

- `<q>` See *Standard assembler syntax fields*.

- `<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF
                D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT
                D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL
                D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector 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.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1 (Armv8.3)

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

64-bit SIMD vector (Q == 0)

VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

128-bit SIMD vector (Q == 1)

if !HaveFCADDExt() then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() & esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;

T1 (Armv8.3)

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | rot | 1 | D | 0 | S | Vn | Vd | 1 | 0 | 0 | 0 | N | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

128-bit SIMD vector (Q == 1)

if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() & esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
Assembler Symbols

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in "S":

<table>
<thead>
<tr>
<th></th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F16</td>
</tr>
<tr>
<td>1</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<rotate> Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in "rot":

<table>
<thead>
<tr>
<th></th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>1</td>
<td>270</td>
</tr>
</tbody>
</table>

Operation

EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    operand3 = D[d+r];
    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]);
        result1 = FPAdd(Elem[operand1,e*2+1,esize],element1,StandardFPSCRValue());
        result2 = FPAdd(Elem[operand1,e*2+1,esize],element3,StandardFPSCRValue());
        Elem[D[d+r],e*2+1,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
**VCEQ (immediate #0)**

Vector Compare Equal to Zero takes each element in a vector, and compares it with zero. If it is equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros. The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

64-bit SIMD vector (Q == 0)

\[
\text{VCEQ}\{<c>\}\{<q>\}.<dt> \{<Dd>,\} <Dm>, \#0
\]

128-bit SIMD vector (Q == 1)

\[
\text{VCEQ}\{<c>\}\{<q>\}.<dt> \{<Dd>,\} <Dm>, \#0
\]

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | D | 1 | 1 | size | 0 | 1 | Vd | F | 0 | 1 | 0 | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

\[
\text{VCEQ}\{<c>\}\{<q>\}.<dt> \{<Dd>,\} <Dm>, \#0
\]

128-bit SIMD vector (Q == 1)

\[
\text{VCEQ}\{<c>\}\{<q>\}.<dt> \{<Dd>,\} <Dm>, \#0
\]

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**CONSTRAINED UNPREDICTABLE behavior**

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operands, encoded in “F:size”:

<table>
<thead>
<tr>
<th>F</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>I8</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>I16</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>I32</td>
<td></td>
</tr>
<tr>
<td>1 01</td>
<td>F16</td>
<td></td>
</tr>
<tr>
<td>1 10</td>
<td>F32</td>
<td></td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareEQ(Elem[D+m+r],e,esize], zero, StandardFPSCRVValue());
            else
                test_passed = (Elem[D+m+r],e,esize] == Zeros(esize));
            Elem[D+d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**VCEQ (register)**

Vector Compare Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

64-bit SIMD vector (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

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

64-bit SIMD vector (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
int_operation = FALSE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 1 | 1 1 1 1 0 | D | size | Vn | | Vd | 1 | 0 | 0 | 0 | N | Q | M | 1 | Vm |
64-bit SIMD vector (Q == 0)

VCEQ{<c>}<q>.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCEQ{<c>}<q>.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
int_operation = FALSE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d =UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VCEQ{<c>}<q>.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCEQ{<c>}<q>.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
int_operation = FALSE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d =UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields.

See Standard assembler syntax fields.

<dt> For encoding A1 and T1: is the data type for the elements of the vectors, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>I8</td>
</tr>
<tr>
<td>01</td>
<td>I16</td>
</tr>
<tr>
<td>10</td>
<td>I32</td>
</tr>
</tbody>
</table>

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if int_operation then
                test_passed = (op1 == op2);
            else
                test_passed = FPCompareEQ(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCGE (immediate #0)

Vector Compare Greater Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is
greater than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set
to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector
elements are fields the same size as the operand vector elements. Depending on settings in the CPACR, NSACR, and HCPTR
registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

\[
\text{VCGE}\{<c>\}{<q>}.<dt> \{<Dd>\}, <Dm>, #0}
\]

128-bit SIMD vector (Q == 1)

\[
\text{VCGE}\{<c>\}{<q>}.<dt> \{<Qd>\}, <Qm>, #0}
\]

T1

```
<table>
<thead>
<tr>
<th>14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>D</td>
</tr>
</tbody>
</table>
```

64-bit SIMD vector (Q == 0)

\[
\text{VCGE}\{<c>\}{<q>}.<dt> \{<Dd>\}, <Dm>, #0}
\]

128-bit SIMD vector (Q == 1)

\[
\text{VCGE}\{<c>\}{<q>}.<dt> \{<Qd>\}, <Qm>, #0}
\]

CONSTRANIED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Assembler Symbols

For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

See *Standard assembler syntax fields*.

Is the data type for the elements of the operands, encoded in “F:size”:

<table>
<thead>
<tr>
<th>F size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 01</td>
<td>F16</td>
</tr>
<tr>
<td>1 10</td>
<td>F32</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```assembly
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regsz-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) >= 0);
            Elem[D[r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Compare Greater Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction VCLE (register).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

**A1**

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

64-bit SIMD vector (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGEtype_unsigned else VCGEtype_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**A2**

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

64-bit SIMD vector (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
vtype = VCGEtype_fp;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | D  | size | Vn | Vd | 0  | 0  | 1  | 1  | N  | Q  | M  | 1  | Vm |
64-bit SIMD vector (Q == 0)

\[
\text{VCGE\{<c>\}\{<q>\}.\{<Dd>, }\{<Dn>, <Dm>}
\]

128-bit SIMD vector (Q == 1)

\[
\text{VCGE\{<c>\}\{<q>\}.\{<Qd>, }\{<Qn>, <Qm>}
\]

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGEtype_fp;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRANDED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

\[
\text{<c> For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.}
\text{For encoding T1 and T2: see Standard assembler syntax fields.}
\]

\[
\text{<q> See Standard assembler syntax fields.}
\]

\[
\text{<dt> For encoding A1 and T1: is the data type for the elements of the operands, encoded in “U:size”:}
\]

\[
\begin{array}{c|c}
\text{U size} & <dt> \\
\hline
0 00 & S8 \\
0 01 & S16 \\
0 10 & S32 \\
1 00 & U8 \\
1 01 & U16 \\
1 10 & U32 \\
\end{array}
\]

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation

```plaintext
enumeration VCGEtype {VCGEtype_signed, VCGEtype_unsigned, VCGEtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
        case vtype of
        when VCGEtype_signed    test_passed = (SInt(op1) >= SInt(op2));
        when VCGEtype_unsigned  test_passed = (UInt(op1) >= UInt(op2));
        when VCGEtype_fp        test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
        Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
VCGT (immediate #0)

Vector Compare Greater Than Zero takes each element in a vector, and compares it with zero. If it is greater than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCGT{<c>{<q}.<dt> {<Dd>,} <Dm}, #0

128-bit SIMD vector (Q == 1)

VCGT{<c>{<q}.<dt> {<Dd>,} <Dm}, #0

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | D  | 1  | 1  | size | 0  | 1  | Vd | F  | 0  | 0  | 0 | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VCGT{<c>{<q}.<dt> {<Dd>,} <Dm}, #0

128-bit SIMD vector (Q == 1)

VCGT{<c>{<q}.<dt> {<Dd>,} <Dm}, #0

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRANDED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Assembler Symbols

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

Is the data type for the elements of the operands, encoded in “F:size”:

<table>
<thead>
<tr>
<th>F</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      if floating_point then
        bits(esize) zero = FPZero('0');
        test_passed = FPCompareGT(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
      else
        test_passed = (SInt(Elem[D[m+r],e,esize]) > 0);
        Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
Vector Compare Greater Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VCLT (register).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

64-bit SIMD vector (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' & & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

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

64-bit SIMD vector (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' & & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & & !HaveFP16Ext() then UNDEFINED;
vtype = VCGTtype_fp;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | U | 1 | 1 | 1 | 1 | 0 | D | size | Vn | Vd | 0 | 0 | 1 | 1 | N | Q | M | 0 | Vm |
64-bit SIMD vector (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T2

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGTtype_fp;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRANGED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1 and T2: see Standard assembler syntax fields.

For encoding A1 and T1: is the data type for the elements of the operands, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>U8</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>U16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>U32</td>
</tr>
</tbody>
</table>

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation

```plaintext
Operation

enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
      case vtype of
        when VCGTtype_signed    test_passed = (SInt(op1) > SInt(op2));
        when VCGTtype_unsigned  test_passed = (UInt(op1) > UInt(op2));
        when VCGTtype_fp        test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
      Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCLE (immediate #0)

Vector Compare Less Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is less than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros. The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPtr registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

128-bit SIMD vector (Q == 1)

VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>Vd</td>
<td>F</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Q</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

128-bit SIMD vector (Q == 1)

VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONstrained UNPREDICTABLE Behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Assembler Symbols

<c>
For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q>
See *Standard assembler syntax fields*.

<dt>
Is the data type for the elements of the operands, encoded in “F:size”:

<table>
<thead>
<tr>
<th>F</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.

<Qm>
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>\*2.

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) <= 0);
                Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Compare Less Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VCGE (register). This means:

- The encodings in this description are named to match the encodings of VCGE (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGE (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

64-bit SIMD vector (Q == 0)

VCLE{<c>{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGE{<c>{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VCLE{<c>{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGE{<c>{<q>}.<dt> <Qd>, <Qm>, <Qn>

A2

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

64-bit SIMD vector (Q == 0)

VCLE{<c>{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGE{<c>{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VCLE{<c>{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGE{<c>{<q>}.<dt> <Qd>, <Qm>, <Qn>

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | U  | 1  | 1  | 1  | 1  | 0  | D  | size | Vn | Vd | 0  | 0  | 1  | 1  | N  | Q  | M  | 1  | Vm |
64-bit SIMD vector (Q == 0)

\[ \text{VCLE} \{<c>\} \{<q>\}. \langle dt \rangle \{<Dd>, <Dn>, <Dm> \}
\]

is equivalent to

\[ \text{VCGE} \{<c>\} \{<q>\}. \langle dt \rangle \{<Dd>, <Dm>, <Dn> \}
\]

128-bit SIMD vector (Q == 1)

\[ \text{VCLE} \{<c>\} \{<q>\}. \langle dt \rangle \{<Qd>, <Qn>, <Qm> \}
\]

is equivalent to

\[ \text{VCGE} \{<c>\} \{<q>\}. \langle dt \rangle \{<Qd>, <Qm>, <Qn> \}
\]

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | D | 0 | sz | Vn | Vd | 1 | 1 | 1 | 0 | N | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

\[ \text{VCLE} \{<c>\} \{<q>\}. \langle dt \rangle \{<Dd>, <Dn>, <Dm> \}
\]

is equivalent to

\[ \text{VCGE} \{<c>\} \{<q>\}. \langle dt \rangle \{<Dd>, <Dm>, <Dn> \}
\]

128-bit SIMD vector (Q == 1)

\[ \text{VCLE} \{<c>\} \{<q>\}. \langle dt \rangle \{<Qd>, <Qn>, <Qm> \}
\]

is equivalent to

\[ \text{VCGE} \{<c>\} \{<q>\}. \langle dt \rangle \{<Qd>, <Qm>, <Qn> \}
\]

Assembler Symbols

- \(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- \(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- \(<Qm>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \(<Qm>\times 2\).
- \(<Qn>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \(<Qn>\times 2\).
- \(<c>\) For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1 and T2: see Standard assembler syntax fields.
- \(<q>\) See Standard assembler syntax fields.
- \(<dt>\) For encoding A1 and T1: is the data type for the elements of the operands, encoded in "U:size":

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>(&lt;dt&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>U8</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>U16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>U32</td>
</tr>
</tbody>
</table>

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>(&lt;dt&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**Operation**

The description of [VCGE (register)](#) gives the operational pseudocode for this instruction.
VCLS

Vector Count Leading Sign Bits counts the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector. The count does not include the topmost bit itself.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit signed integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>

128-bit SIMD vector (Q == 1)

VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 1 1 1 1 1 1 1 1 1 | D 1 1 | size | 0 0 | Vd | 0 1 0 0 0 | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>

128-bit SIMD vector (Q == 1)

VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operands, encoded in “size”:
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingSignBits(Elem[D[m+r],e,esize])<esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCLT (immediate #0)

Vector Compare Less Than Zero takes each element in a vector, and compares it with zero. If it is less than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>Vd</td>
<td></td>
<td>F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>}, {<Dm>}, #0

128-bit SIMD vector (Q == 1)

VCLT{<c>}{<q>}.<dt> {<Dd>}, {<Qm>}, #0

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
float point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>0</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>Vd</td>
<td></td>
<td>F</td>
<td>1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>}, {<Dm>}, #0

128-bit SIMD vector (Q == 1)

VCLT{<c>}{<q>}.<dt> {<Dd>}, {<Qm>}, #0

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
float point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operands, encoded in “F:size”:

<table>
<thead>
<tr>
<th>F size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 01</td>
<td>F16</td>
</tr>
<tr>
<td>1 10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGT(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) < 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

VCLT (immediate #0)
VCLT (register)

Vector Compare Less Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VCGT (register). This means:

- The encodings in this description are named to match the encodings of VCGT (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGT (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

A2

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

64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|----------------|
| 1 1 1 | U | 1 | 1 | 1 | 1 | 0 | D | size | Vn | Vd | 0 | 0 | 1 | 1 | N | Q | M | 0 | Vm |
```
64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 1 1 1 1 0 | D 1 sz | Vn | Vd | 1 1 1 0 | N Q M 0 | Vm |

64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

Assembler Symbols

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<c> For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1 and T2: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> For encoding A1 and T1: is the data type for the elements of the operands, encoded in "U:size":

<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
</tbody>
</table>

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**Operation**

The description of [VCGT (register)](#) gives the operational pseudocode for this instruction.

---

Internal version only: isa v01_19, pseudocode v2020-09.xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector. The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The result vector elements are the same data type as the operand vector elements. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCLZ{<c}>{<q>}.<dt> <Qd>, <Qm>

128-bit SIMD vector (Q == 1)

VCLZ{<c}>{<q>}.<dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' & & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 | D | 1 1 | size | 0 0 | Vd | 0 | 1 | 0 | 0 | 1 | Q | M | 0 | Vm

64-bit SIMD vector (Q == 0)

VCLZ{<c}>{<q>}.<dt> <Qd>, <Qm>

128-bit SIMD vector (Q == 1)

VCLZ{<c}>{<q>}.<dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' & & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operands, encoded in "size":

VCLZ
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>18</td>
</tr>
<tr>
<td>01</td>
<td>116</td>
</tr>
<tr>
<td>10</td>
<td>132</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D.Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M.Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D.Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M.Vm" field.

**Operation**

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      Elem[D[d+r],e,esize] = CountLeadingZeroBits(Elem[D[m+r],e,esize])<esize-1:0>;

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector 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. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1
(Armv8.3)

```
1 1 1 1 1 1 0 | rot | D | 1 | S | Vn | Vd | 1 | 0 | 0 | 0 | N | Q | M | 0 | Vm
64-bit SIMD vector (Q == 0)
```

```
VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```

### 128-bit SIMD vector (Q == 1)

```
VCMLA{<q>}.<dt> <Dq>, <Dn>, <Qm>, #<rotate>
```

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

### T1
(Armv8.3)

```
1 1 1 1 1 1 0 | rot | D | 1 | S | Vn | Vd | 1 | 0 | 0 | 0 | N | Q | M | 0 | Vm
```
64-bit SIMD vector (Q == 0)

VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

128-bit SIMD vector (Q == 1)

VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;

Assembler Symbols

<q> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the vectors, encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F16</td>
</tr>
<tr>
<td>1</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<rotate> Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in “rot”:

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>180</td>
</tr>
<tr>
<td>11</td>
<td>270</td>
</tr>
</tbody>
</table>
Operation

EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    operand3 = D[d+r];
    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+1,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 = FPNeg(Elem[operand1,e*2+1,esize]);
        result1 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, StandardFPSCRValue());
        result2 = FPMulAdd(Elem[operand3,e*2+1,esize],element4,element3, StandardFPSCRValue());
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
VCMLA (by element)

Vector 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. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1 (Armv8.3)

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

64-bit SIMD vector of half-precision floating-point (S == 0 && Q == 0)

VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>

64-bit SIMD vector of single-precision floating-point (S == 1 && Q == 0)

VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>

128-bit SIMD vector of half-precision floating-point (S == 0 && Q == 1)

VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>

128-bit SIMD vector of single-precision floating-point (S == 1 && Q == 1)

VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>

if !HaveFCADEExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
index = if S=='1' then 0 else UInt(M);

T1 (Armv8.3)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  1 |  1 |  1 |  1 |  1 |  1 |  1 |  0 | S | D | rot | Vn | Vd | 1 | 0 | 0 | 0 | N | Q | M | 0 | Vm |
64-bit SIMD vector of half-precision floating-point (S == 0 && Q == 0)

VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>

64-bit SIMD vector of single-precision floating-point (S == 1 && Q == 0)

VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>

128-bit SIMD vector of half-precision floating-point (S == 0 && Q == 1)

VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>

128-bit SIMD vector of single-precision floating-point (S == 1 && Q == 1)

VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>

if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
index = if S=='1' then 0 else UInt(M);

Assembler Symbols

<q> See Standard assembler syntax fields.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> For the half-precision scalar variant: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field. For the single-precision scalar variant: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

<rotate> Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>180</td>
</tr>
<tr>
<td>11</td>
<td>270</td>
</tr>
</tbody>
</table>
Operation

EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
  operand1 = D[n+r];
  operand2 = Din[m];
  operand3 = D[d+r];
  for e = 0 to (elements DIV 2)-1
    case rot of
      when '00'
        element1 = Elem[operand2,index*2,esize];
        element2 = Elem[operand1,e*2,esize];
        element3 = Elem[operand2,index*2+1,esize];
        element4 = Elem[operand1,e*2,esize];
      when '01'
        element1 = FPNeg(Elem[operand2,index*2+1,esize]);
        element2 = Elem[operand1,e*2+1,esize];
        element3 = Elem[operand2,index*2,esize];
        element4 = Elem[operand1,e*2,esize];
      when '10'
        element1 = FPNeg(Elem[operand2,index*2,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 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, StandardFPSCRValue());
    result1 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, StandardFPSCRValue());
    result2 = FPMulAdd(Elem[operand3,e*2+1,esize],element4,element3, StandardFPSCRValue());
    Elem[D[d+r],e*2,esize] = result1;
    Elem[D[d+r],e*2+1,esize] = result2;
VCMP

Vector Compare compares two floating-point registers, or one floating-point register and zero. It writes the result to the **FPSCR** flags. These are normally transferred to the **PSTATE** {N, Z, C, V} Condition flags by a subsequent VMRS instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

**A1**

<table>
<thead>
<tr>
<th><img src="image" alt="Binary representation" /></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cond</strong></td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

Half-precision scalar (size == 01)  
(Armv8.2)

VCMP{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, <Dm>

**CONSTRANDED UNPREDICTABLE behavior**

If **size == '01' && cond != '1110'**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**A2**

<table>
<thead>
<tr>
<th><img src="image" alt="Binary representation" /></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cond</strong></td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

Constrained UNPREDICTABLE behavior
Half-precision scalar (size == 01)  
(Armv8.2)

VCMP{<c>}{<q>}.F16 <Sd>, #0.0

Single-precision scalar (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, #0.0

Double-precision scalar (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, #0.0

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | size | 0 | 1 | M | 0 | Vm |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |   |   |   |   |

Half-precision scalar (size == 01)  
(Armv8.2)

VCMP{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Half-precision scalar (size == 01)
(Armv8.2)

VCMP{<c>}{<q>}.F16 <Sd>, #0.0

Single-precision scalar (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, #0.0

Double-precision scalar (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, #0.0

if size == '00' || (size == '01' && ! HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D);
    when '10' esize = 32; d = UInt(Vd:D);
    when '11' esize = 64; d = UInt(D:Vd);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

See Standard assembler syntax fields.

<Sm> is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> is the 32-bit name of the SIMD&FP source register, encoded in the "Vd:D" field.
<Sm> is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sm> is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    bits(4) nzcv;
    case esize of
        when 16
            bits(16) op16 = if with_zero then FPZero('0') else S[m]<15:0>;
            nzcv = FPCompare(S[d]<15:0>, op16, quiet_nan_exc, FPSCR[]);
        when 32
            bits(32) op32 = if with_zero then FPZero('0') else S[m];
            nzcv = FPCompare(S[d], op32, quiet_nan_exc, FPSCR[]);
        when 64
            bits(64) op64 = if with_zero then FPZero('0') else D[m];
            nzcv = FPCompare(D[d], op64, quiet_nan_exc, FPSCR[]);
    end_case
    FPSCR<31:28> = nzcv; // FPSCR.<N,Z,C,V> set to nzcv
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the FPSCR condition flags to N=0, Z=0, C=1, and V=1. If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Compare, raising Invalid Operation on NaN compares two floating-point registers, or one floating-point register and zero. It writes the result to the FPSCR flags. These are normally transferred to the PSTATE.\{N, Z, C, V\} Condition flags by a subsequent VMRS instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

Depending on settings in the CPACR, NSACR, and HCPTTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 ( A1 and A2 ) and T32 ( T1 and T2 ).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: |
| != 1111           | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 0 | 1 | 0 | 0 | Vd | 1 | 0 | size | 1 | 1 | M | 0 | Vm |
| cond              | E |

Half-precision scalar (size == 01)
(Armv8.2)

VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet nan exc = (E == '1'); with_zero = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: | :-----------------: |
| != 1111           | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 0 | 1 | 0 | 1 | Vd | 1 | 0 | size | 1 | 1 | 0(0) | 0(0) | 0(0) | 0(0) |
| cond              | E |
Half-precision scalar (size == 01)
(Armv8.2)

VCMPE{<c>}{<q>}.F16 <Sd>, #0.0

Single-precision scalar (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, #0.0

Double-precision scalar (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, #0.0

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size | 1 | 1 |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>Vm</td>
<td></td>
</tr>
</tbody>
</table>

E

Half-precision scalar (size == 01)
(Armv8.2)

VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Half-precision scalar (size == 01)  
(Armv8.2)

\[ \text{VCMPE}\{<c>\}\{<q>\}\cdot\text{F16} \ <\text{Sd}\}, \ #0.0 \]

Single-precision scalar (size == 10)

\[ \text{VCMPE}\{<c>\}\{<q>\}\cdot\text{F32} \ <\text{Sd}\}, \ #0.0 \]

Double-precision scalar (size == 11)

\[ \text{VCMPE}\{<c>\}\{<q>\}\cdot\text{F64} \ <\text{Dd}\}, \ #0.0 \]

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation

if **ConditionPassed**() then
    EncodingSpecificOperations(); **CheckVFPEnabled**(TRUE);
    bits(4) nzcv;
    case esize of
        when 16
            bits(16) op16 = if with_zero then **FPZero**(‘0’) else $s[m]<15:0>$;
            nzcv = **FPCompare**(S[d]<15:0>, op16, quiet_nan_exc, FPSCR[]);
        when 32
            bits(32) op32 = if with_zero then **FPZero**(‘0’) else $s[m]$;
            nzcv = **FPCompare**(S[d], op32, quiet_nan_exc, FPSCR[]);
        when 64
            bits(64) op64 = if with_zero then **FPZero**(‘0’) else $d[m]$;
            nzcv = **FPCompare**(D[d], op64, quiet_nan_exc, FPSCR[]);
    endcase;
    FPSCR<31:28> = nzcv; // FPSCR.<N,Z,C,V> set to nzcv

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the **FPSCR** condition flags to N=0, Z=0, C=1, and V=1. If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Count Set Bits counts the number of bits that are one in each element in a vector, and places the results in a second vector.
The operand vector elements must be 8-bit fields.
The result vector elements are 8-bit integers.
Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCNT{<c>{<q>}.8 <Dd>, <Dm> // (Encoded as Q = 0)

128-bit SIMD vector (Q == 1)

VCNT{<c>{<q>}.8 <Qd>, <Qm> // (Encoded as Q = 1)

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8; elements = 8;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VCNT{<c>{<q>}.8 <Dd>, <Dm> // (Encoded as Q = 0)

128-bit SIMD vector (Q == 1)

VCNT{<c>{<q>}.8 <Qd>, <Qm> // (Encoded as Q = 1)

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8; elements = 8;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation

if `ConditionPassed()` then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for `r` = 0 to `regs-1`
        for `e` = 0 to `elements-1`
            `Elem[0][d+r],e,esize] = BitCount(Elem[0][m+r],e,esize]<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
VCVT (from single-precision to BFloat16, Advanced SIMD)

Vector Convert from single-precision to BFloat16 converts each 32-bit element in a vector from single-precision floating-point to BFloat16 format, and writes the result into a second vector. The result vector elements are half the width of the source vector elements.

Unlike the BFloat16 multiplication instructions, this instruction uses the Round to Nearest rounding mode, and can generate a floating-point exception that causes cumulative exception bits in the **FPSCR** to be set.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1** (Armv8.6)

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

```
A1

VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<@> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer m = UInt(M:Vm);
```

**T1** (Armv8.6)

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  |
| Vd | 0  | 1  | 1  | 0  | 1  | M  | 0  | Vm |
```

```
T1

VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<@> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer m = UInt(M:Vm);
```

**Assembler Symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  For encoding T1: see *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>`*2.
Operation

bits(128) operand;
bits(64) result;

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    operand = Q[m>>1];
    for e = 0 to 3
        bits(32) op = Elem[operand, e, 32];
        Elem[result, e, 16] = FPConvertBF(op, StandardFPSCRValue());
    D[d] = result;
VCVT (between double-precision and single-precision)

Convert between double-precision and single-precision does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
! = 1111  | 1 1 1 0 1 | D 1 1 0 1 1 1 | Vd  | 1 0 1 x 1 1 | M 0   | Vm
cond     | size
```

**Single-precision to double-precision (size == 10)**

VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>

**Double-precision to single-precision (size == 11)**

VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>

```python
double_to_single = (size == '11');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

### T1

```
1 1 1 0 1 1 1 0 1 | D 1 1 0 1 1 1 | Vd  | 1 0 1 x 1 1 | M 0   | Vm
size
```

**Single-precision to double-precision (size == 10)**

VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>

**Double-precision to single-precision (size == 11)**

VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>

```python
double_to_single = (size == '11');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if double_to_single then
        S[d] = FPConvert(D[m], FPSCR[]);
    else
        D[d] = FPConvert(S[m], FPSCR[]);

VCVT (between half-precision and single-precision, Advanced SIMD)

Vector Convert between half-precision and single-precision converts each element in a vector from single-precision to half-precision floating-point, or from half-precision to single-precision, and places the results in a second vector. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 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 D 1 1 size 1 0   Vd 0 1 1 op 0 0 M 0 Vm
```

Half-precision to single-precision (op == 1)

```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm>  // (Encoded as op = 1)
```

Single-precision to half-precision (op == 0)

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm>  // (Encoded as op = 0)
```

if size != '01' then UNDEFINED;
if half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16; elements = 4;
m = UInt(M:Vm); d = UInt(D:Vd);

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 D 1 1 size 1 0   Vd 0 1 1 op 0 0 M 0 Vm
```

Half-precision to single-precision (op == 1)

```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm>  // (Encoded as op = 1)
```

Single-precision to half-precision (op == 0)

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm>  // (Encoded as op = 0)
```

if size != '01' then UNDEFINED;
if half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16; elements = 4;
m = UInt(M:Vm); d = UInt(D:Vd);

Assembler Symbols

<c>  For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd>  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<q>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

### Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
for e = 0 to elements-1
  if half_to_single then
    Elem[d][Q[d>>1],e,32] = FPConvert(Elem[Din][m],e,16), StandardFPSCRValue();
  else
    Elem[D[d],e,16] = FPConvert(Elem[Qin][m>>1],e,32), StandardFPSCRValue();

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCVT (between floating-point and integer, Advanced SIMD)

Vector Convert between floating-point and integer converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;

to_integer = (op<1> == '1'); unsigned = (op<0> == '1');

case size of
  when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
    d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 1 1 1 1 1 1 1 1 1 1 1 1 | D 1 1 | size 1 1 | Vd 0 1 1 | op Q M 0 | Vm |

64-bit SIMD vector (Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
to_integer = (op<1> == '1'); unsigned = (op<0> == '1');

case size of
  when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
    d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt1> Is the data type for the elements of the destination vector, encoded in “size:op”:

<table>
<thead>
<tr>
<th>size</th>
<th>op</th>
<th>&lt;dt1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0x</td>
<td>F16</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>S16</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>U16</td>
</tr>
<tr>
<td>10</td>
<td>0x</td>
<td>F32</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>U32</td>
</tr>
</tbody>
</table>

<dt2> Is the data type for the elements of the source vector, encoded in “size:op”:

<table>
<thead>
<tr>
<th>size</th>
<th>op</th>
<th>&lt;dt2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00</td>
<td>S16</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>U16</td>
</tr>
<tr>
<td>01</td>
<td>1x</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>S32</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>U32</td>
</tr>
<tr>
<td>10</td>
<td>1x</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDEnabled();
   bits(esize) result;
   for r = 0 to regs-1
      for e = 0 to elements-1
         op1 = Elem[D][m+r],e,esize];
         if to_integer then
            result = FPToFixed(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_ZERO);
         else
            result = FixedToFP(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_TIEEVEN);
         Elem[D][d+r],e,esize] = result;

VCVT (between floating-point and integer, Advanced SIMD)
VCVT (floating-point to integer, floating-point)

Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in a second register. 

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
<table>
<thead>
<tr>
<th>31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>
```

VCVT (floating-point to integer, floating-point)
Half-precision scalar (opc2 == 100 && size == 01) (Armv8.2)

VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Half-precision scalar (opc2 == 101 && size == 01) (Armv8.2)

VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar (opc2 == 100 && size == 10)

VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar (opc2 == 101 && size == 10)

VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar (opc2 == 100 && size == 11)

VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Double-precision scalar (opc2 == 101 && size == 11)

VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>

if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
  unsigned = (opc2<0> == '0');
  rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
  d = UInt(Vd:D);
  case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
else
  unsigned = (op == '0');
  rounding = FPRoundingMode(FPSCR[]);
  m = UInt(Vm:M);
  case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);
end

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 1 0 1</td>
<td>D</td>
</tr>
</tbody>
</table>

VCVT (floating-point to integer, floating-point)
Half-precision scalar (opc2 == 100 && size == 01)  
(Armv8.2)

VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Half-precision scalar (opc2 == 101 && size == 01)  
(Armv8.2)

VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar (opc2 == 100 && size == 10)

VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar (opc2 == 101 && size == 10)

VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar (opc2 == 100 && size == 11)

VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Double-precision scalar (opc2 == 101 && size == 11)

VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>

if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
    else
        unsigned = (op == '0');
        rounding = FPRoundingMode(FPSCR[]);
        m = UInt(Vm:M);
        case size of
            when '01' esize = 16; d = UInt(Vd:D);
            when '10' esize = 32; d = UInt(Vd:D);
            when '11' esize = 64; d = UInt(D:Vd);

CON crunched UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Floating-point data-processing for the T32 instruction set, or Floating-point data-processing for the A32 instruction set.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    if to integer then
        case esize of
            when 16
                S[d] = FPToF(x;\texttt{Fixed}S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
            when 32
                S[d] = FPToF(x;\texttt{Fixed}S[m], 0, unsigned, FPSCR[], rounding);
            when 64
                D[d] = FPToF(x;\texttt{Fixed}D[m], 0, unsigned, FPSCR[], rounding);
        else
            case esize of
                when 16
                    bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
                    S[d] = Zeros(16):fp16;
                when 32
                    S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
                when 64
                    D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
```

VCVT (floating-point to integer, floating-point)
VCVT (integer to floating-point, floating-point)

Convert integer to floating-point converts a 32-bit integer to floating-point using the rounding mode specified by the FPSCR, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111| 1  | 1  | 1  | 1  | 0  | 1  | D  | 1  | 1  | 1  | 0  | 0  | 0  | Vd | 1  | 0  | size | op  | 1 | M  | 0  | Vm |

cond | opc2

Half-precision scalar (size == 01)

(ARMv8.2)

VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>

if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to integer = (opc2<2> == '1');
if to integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
    else
        unsigned = (op == '0');
        rounding = FPRoundingMode(FPSCR[]);
        m = UInt(Vm:M);
        case size of
            when '01' esize = 16; d = UInt(Vd:D);
            when '10' esize = 32; d = UInt(Vd:D);
            when '11' esize = 64; d = UInt(D:Vd);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1
VCVT{<c>}{<q>}.F16.<dt>  <Sd>,  <Sm>

VCVT{<c>}{<q>}.F32.<dt>  <Sd>,  <Sm>

VCVT{<c>}{<q>}.F64.<dt>  <Dd>,  <Sm>

if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
to integer = (opc2<2> == '1');
if to integer then
  unsigned = (opc2<0> == '0');
  rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
d = UInt(Vd:D);
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
else
  unsigned = (op == '0');
  rounding = FPRoundingMode(FPSCR[]);
m = UInt(Vm:M);
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Floating-point data-processing for the T32 instruction set, or Floating-point data-processing for the A32 instruction set.

Assembler Symbols

<c>  See Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<dt>  Is the data type for the operand, encoded in "op"

<table>
<thead>
<tr>
<th>op</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U32</td>
</tr>
<tr>
<td>1</td>
<td>S32</td>
</tr>
</tbody>
</table>

<Sd>  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Sm>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if to_integer then
    case esize of
        when 16
            $[d] = FPToFixed($[m]<15:0>, 0, unsigned, FPSCR[], rounding);
        when 32
            $[d] = FPToFixed($[m], 0, unsigned, FPSCR[], rounding);
        when 64
            $[d] = FPToFixed($[m], 0, unsigned, FPSCR[], rounding);
    else
        case esize of
            when 16
                bits(16) fp16 = FixedToFP($[m], 0, unsigned, FPSCR[], rounding);
                $[d] = Zeros(16):fp16;
            when 32
                $[d] = FixedToFP($[m], 0, unsigned, FPSCR[], rounding);
            when 64
                $[d] = FixedToFP($[m], 0, unsigned, FPSCR[], rounding);
Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCVT (between floating-point and fixed-point, Advanced SIMD)

Vector Convert between floating-point and fixed-point converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector. The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```assembly
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 1 1 0 0 1 1 U 1 D imm6 Vd 1 1 op 0 Q M 1 Vm
```

64-bit SIMD vector (imm6 != 000xxx && Q == 0)

```
VCVT{c}{q}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

128-bit SIMD vector (imm6 != 000xxx && Q == 1)

```
VCVT{c}{q}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```

if imm6 == '000000' then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 == '10xxxx' then UNDEFINED;
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op<0> == '1'); frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
case op<1> of
  when '0' esize = 16; elements = 4;
  when '1' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

```assembly
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 U 1 1 1 1 1 1 D imm6 Vd 1 1 op 0 Q M 1 Vm
```
64-bit SIMD vector (imm6 != 000xxx && Q == 0)

VCVT{<c>}{<q>}.<dt1> <Dd>, <Dm>, #<fbits>

128-bit SIMD vector (imm6 != 000xxx && Q == 1)

VCVT{<c>}{<q>}.<dt2> <Qd>, <Qm>, #<fbits>

if imm6 == '000xxx' then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 == '10xxxx' then UNDEFINED;
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op<0> == '1');
frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
case op<1>
when '0' esize = 16; elements = 4;
when '1' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
end

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<q> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<dt1> Is the data type for the elements of the destination vector, encoded in “op:U”:

<table>
<thead>
<tr>
<th>op</th>
<th>U</th>
<th>&lt;dt1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>F16</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>S16</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>U16</td>
</tr>
<tr>
<td>10</td>
<td>x</td>
<td>F32</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>S32</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>U32</td>
</tr>
</tbody>
</table>

<dt2> Is the data type for the elements of the source vector, encoded in “op:U”:

<table>
<thead>
<tr>
<th>op</th>
<th>U</th>
<th>&lt;dt2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>S16</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>U16</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>S32</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>U32</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<fbits> The number of fraction bits in the fixed point number, in the range 1 to 32 for 32-bit elements, or in the range 1 to 16 for 16-bit elements:

- (64 - <fbits>) is encoded in imm6.

An assembler can permit an <fbits> value of 0. This is encoded as floating-point to integer or integer to floating-point instruction, see VCVT (between floating-point and integer, Advanced SIMD).
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            opl = Elem[D[m+r],e,esize];
            if to_fixed then
                result = FPToFixed(opl, frac_bits, unsigned, StandardFPSCRValue(),
                                  FPRounding_ZERO);
            else
                result = FixedToFP(opl, frac_bits, unsigned, StandardFPSCRValue(),
                                    FPRounding_TIEEVEN);
            Elem[D[d+r],e,esize] = result;

VCVT (between floating-point and fixed-point, Advanced SIMD)
VCVT (between floating-point and fixed-point, floating-point)

Convert between floating-point and fixed-point converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point. Software can specify the fixed-point value as either signed or unsigned. The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != 1111 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 1 | op | 1 | U | Vd | 1 | 0 | sf | sx | 1 | i | 0 | imm4 |

- **cond**
- **Half-precision scalar (op == 0 && sf == 01)**
  (Armv8.2)
  
  \[ \text{VCVT}\{c}\{q}\}.F16.<dt> <Sdm>, <Sdm>, #<fbits> \]

- **Half-precision scalar (op == 1 && sf == 01)**
  (Armv8.2)
  
  \[ \text{VCVT}\{c}\{q}\}.<dt>.F16 <Sdm>, <Sdm>, #<fbits> \]

- **Single-precision scalar (op == 0 && sf == 10)**
  
  \[ \text{VCVT}\{c}\{q}\}.<dt>.F32 <Sdm>, <Sdm>, #<fbits> \]

- **Single-precision scalar (op == 1 && sf == 10)**
  
  \[ \text{VCVT}\{c}\{q}\}.<dt>.F32 <Sdm>, <Sdm>, #<fbits> \]

- **Double-precision scalar (op == 0 && sf == 11)**
  
  \[ \text{VCVT}\{c}\{q}\}.<dt>.F64 <Ddm>, <Ddm>, #<fbits> \]

- **Double-precision scalar (op == 1 && sf == 11)**
  
  \[ \text{VCVT}\{c}\{q}\}.<dt>.F64 <Ddm>, <Ddm>, #<fbits> \]

if sf == '00' || (sf == '01' && !HaveFP16Ext()) then UNDEFINED;
if sf == '01' && cond != '1110' then UNPREDICTABLE;
to_fixed = (op == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
case sf of
  when '01' fp_size = 16; d = UInt(Vd:D);
  when '10' fp_size = 32; d = UInt(Vd:D);
  when '11' fp_size = 64; d = UInt(D:Vd);
if frac_bits < 0 then UNPREDICTABLE;
CONSTRANGED UNPREDICTABLE behavior

If \( \text{frac\_bits} < 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

**T1**

| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | op | 1 | U | Vd | 1 | 0 | sf | sx | 1 | i | 0 | imm4 |

**Half-precision scalar (op == 0 \&\& sf == 01)**
(Armv8.2)

\[ \text{VCVT}\{\langle c\rangle\}\{\langle q\rangle\}.\text{F16}.<\text{dt}> <\text{Sdm}>, <\text{Sdm}>, \#<\text{fbits}> \]

**Half-precision scalar (op == 1 \&\& sf == 01)**
(Armv8.2)

\[ \text{VCVT}\{\langle c\rangle\}\{\langle q\rangle\}.\text{F16} <\text{Sdm}>, <\text{Sdm}>, \#<\text{fbits}> \]

**Single-precision scalar (op == 0 \&\& sf == 10)**

\[ \text{VCVT}\{\langle c\rangle\}\{\langle q\rangle\}.\text{F32}.<\text{dt}> <\text{Sdm}>, <\text{Sdm}>, \#<\text{fbits}> \]

**Single-precision scalar (op == 1 \&\& sf == 10)**

\[ \text{VCVT}\{\langle c\rangle\}\{\langle q\rangle\}.\text{F32} <\text{Sdm}>, <\text{Sdm}>, \#<\text{fbits}> \]

**Double-precision scalar (op == 0 \&\& sf == 11)**

\[ \text{VCVT}\{\langle c\rangle\}\{\langle q\rangle\}.\text{F64}.<\text{dt}> <\text{Ddm}>, <\text{Ddm}>, \#<\text{fbits}> \]

**Double-precision scalar (op == 1 \&\& sf == 11)**

\[ \text{VCVT}\{\langle c\rangle\}\{\langle q\rangle\}.\text{F64} <\text{Ddm}>, <\text{Ddm}>, \#<\text{fbits}> \]

```c
if sf == '00' || (sf == '01' \&\& !\text{HaveFP16Ext}()) then UNDEFINED;
if sf == '01' \&\& \text{InITBlock}() then UNPREDICTABLE;
to\_fixed = (op == '1'); \text{unsigned} = (U == '1');
size = if sx == '0' then 16 else 32;
frac\_bits = size - \text{UInt}(imm4:i);
case sf of
  when '01' fp\_size = 16; d = \text{UInt}(Vd:D);
  when '10' fp\_size = 32; d = \text{UInt}(Vd:D);
  when '11' fp\_size = 64; d = \text{UInt}(D:Vd);
if frac\_bits < 0 then UNPREDICTABLE;
```

CONSTRANGED UNPREDICTABLE behavior

If \( \text{frac\_bits} < 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VCVT (between floating-point and fixed-point)*.
Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the fixed-point number, encoded in "U: sx":

<table>
<thead>
<tr>
<th>U</th>
<th>sx</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>S32</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>U16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>U32</td>
</tr>
</tbody>
</table>

<Sdm> Is the 32-bit name of the SIMD&FP destination and source register, encoded in the "V:d:D" field.

<Ddm> Is the 64-bit name of the SIMD&FP destination and source register, encoded in the "D:V:d" field.

<fbits> The number of fraction bits in the fixed-point number:

- If <dt> is S16 or U16, <fbits> must be in the range 0-16. (16 - <fbits>) is encoded in [imm4, i]
- If <dt> is S32 or U32, <fbits> must be in the range 1-32. (32 - <fbits>) is encoded in [imm4, i].

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
else
    case fp_size of
        when 16
            bits(size) result;
            case fp_size of
                when 16
                    result = FPToFixed($d<15:0>, frac_bits, unsigned, FPSCR[], FPRounding_ZERO);
                    $d = Extend(result, 32, unsigned);
                when 32
                    result = FPToFixed($d, frac_bits, unsigned, FPSCR[], FPRounding_ZERO);
                    $d = Extend(result, 32, unsigned);
                when 64
                    result = FPToFixed(D[d], frac_bits, unsigned, FPSCR[], FPRounding_ZERO);
                    D[d] = Extend(result, 64, unsigned);
        when 32
            S[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding_TIEEVEN);
        when 64
            D[d] = FixedToFP(D[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding_TIEEVEN);

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCVTA (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest with Ties to Away converts each element in a vector from floating-point to integer using the Round to Nearest with Ties to Away rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.
The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.
Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode.
For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>

128-bit SIMD vector (Q == 1)

VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | D  | 1  | 1  | 1  | 1  | Vd | 0  | 0  | 0  | 0  | op | Q | M | 0  | Vm |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

64-bit SIMD vector (Q == 0)

VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>

128-bit SIMD vector (Q == 1)

VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
CONSTRAINED UNPREDICTABLE behavior

If \texttt{InITBlock()}, then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- \texttt{<q>} See \textit{Standard assembler syntax fields}.
- \texttt{<dt>} Is the data type for the elements of the destination, encoded in “\texttt{op:size}”:
  
<table>
<thead>
<tr>
<th>\texttt{op}</th>
<th>\texttt{size}</th>
<th>\texttt{&lt;dt&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>U16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>U32</td>
</tr>
</tbody>
</table>

- \texttt{<dt2>} Is the data type for the elements of the source vector, encoded in “\texttt{size}”:
  
<table>
<thead>
<tr>
<th>\texttt{size}</th>
<th>\texttt{&lt;dt2&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

- \texttt{<Qd>} Is the 128-bit name of the SIMD&FP destination register, encoded in the "D.Vd" field as \texttt{<Qd>*2}.
- \texttt{<Qm>} Is the 128-bit name of the SIMD&FP source register, encoded in the "M.Vm" field as \texttt{<Qm>*2}.
- \texttt{<Dd>} Is the 64-bit name of the SIMD&FP destination register, encoded in the "D.Vd" field.
- \texttt{<Dm>} Is the 64-bit name of the SIMD&FP source register, encoded in the "M.Vm" field.

Operation

\begin{verbatim}
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
  for e = 0 to elements-1
    Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                      StandardFPSCRValue(), rounding);
\end{verbatim}

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCVTA (floating-point)

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPtr, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
1 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 1 | 1 | 0 | 0 | Vd | 1 | 0 | != 00 | op | 1 | M | 0 | Vm

RM  size
```

Half-precision scalar (size == 01) (Armv8.2)

VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPD::decodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
```

```
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
```

T1

```
1 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 1 | 1 | 0 | 0 | Vd | 1 | 0 | != 00 | op | 1 | M | 0 | Vm

RM  size
```
Half-precision scalar (size == 01)
(Armv8.2)

VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);

CONstrained UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the destination, encoded in "op":

<table>
<thead>
<tr>
<th>op</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U32</td>
</tr>
<tr>
<td>1</td>
<td>S32</td>
</tr>
</tbody>
</table>

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);

VCVTB

Convert to or from a half-precision value in the bottom half of a single-precision register does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------------------------------------------|----------|
| != 1111 1 1 1 0 1 | D 1 1 0 0 1 | op | Vd | 1 0 1 | sz | 0 1 | M | 0 | Vm |

**cond** | T

**Half-precision to single-precision (op == 0 & sz == 0)**

VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>

**Half-precision to double-precision (op == 0 & sz == 1)**

VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>

**Single-precision to half-precision (op == 1 & sz == 0)**

VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>

**Double-precision to half-precision (op == 1 & sz == 1)**

VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>

```plaintext
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
  if convert_from_half then
    d = UInt(D:Vd); m = UInt(Vm:M);
  else
    d = UInt(Vd:D); m = UInt(M:Vm);
else
  d = UInt(Vd:D); m = UInt(Vm:M);
```

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|-----------------------------------------|----------|
| 1 1 1 0 1 1 1 0 0 1 | D 1 1 0 0 1 | op | Vd | 1 0 1 | sz | 0 1 | M | 0 | Vm |

T
Half-precision to single-precision (\(op == 0 \&\& sz == 0\))

\[
\text{VCVTB}\{<c>\}{<q>}.F32.F16 \quad <Sd>, \quad <Sm>
\]

Half-precision to double-precision (\(op == 0 \&\& sz == 1\))

\[
\text{VCVTB}\{<c>\}{<q>}.F64.F16 \quad <Dd>, \quad <Sm>
\]

Single-precision to half-precision (\(op == 1 \&\& sz == 0\))

\[
\text{VCVTB}\{<c>\}{<q>}.F16.F32 \quad <Sd>, \quad <Sm>
\]

Double-precision to half-precision (\(op == 1 \&\& sz == 1\))

\[
\text{VCVTB}\{<c>\}{<q>}.F16.F64 \quad <Sd>, \quad <Dm>
\]

\[
\begin{align*}
\text{uses_double} &= \text{(sz == '1')} \land \text{convert_from_half} = \text{(op == '0')} \\
\text{lowbit} &= (\text{if T == '1' then 16 else 0}) \\
\text{if uses_double then} \\
\text{if convert_from_half then} \\
\quad d &= \text{UInt}(D:Vd); \quad m = \text{UInt}(Vm:M); \\
\text{else} \\
\quad d &= \text{UInt}(Vd:D); \quad m = \text{UInt}(M:Vm); \\
\text{else} \\
\quad d &= \text{UInt}(Vd:D); \quad m = \text{UInt}(Vm:M);
\end{align*}
\]

Assembler Symbols

- `<c>` See *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

\[
\begin{align*}
\text{if } \text{ConditionPassed}() \text{ then} \\
\text{if } \text{ConversionSpecificOperations}(); \quad \text{CheckVFPEnabled}(\text{TRUE}); \\
\text{bits(16)} \quad \text{hp}; \\
\text{if convert_from_half then} \\
\quad \text{hp} &= \text{S}[m]\text{lowbit+15:lowbit}; \\
\text{if uses_double then} \\
\quad D[d] &= \text{FPConvert}(\text{hp}, \text{FPSCR[]}); \\
\text{else} \\
\quad S[d] &= \text{FPConvert}(\text{hp}, \text{FPSCR[]}); \\
\text{else} \\
\text{if uses_double then} \\
\quad \text{hp} &= \text{FPConvert}(D[m], \text{FPSCR[]}); \\
\text{else} \\
\quad \text{hp} &= \text{FPConvert}(S[m], \text{FPSCR[]}); \\
\quad S[d]\text{lowbit+15:lowbit} &= \text{hp};
\end{align*}
\]
VCVTB (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result into the bottom half of a single precision register, preserving the top 16 bits of the destination register. Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the FPSCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the FPSCR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPSCR.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1 (Armv8.6)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 1111</td>
</tr>
</tbody>
</table>

cond

T1 (Armv8.6)

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 1 0 1</td>
</tr>
</tbody>
</table>

Asmber Symbols

<cond> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnable(TRUE);

    S[d]<15:0> = FPConvertBF(S[m], FPSCR[]);
VCVTM (Advanced SIMD)

Vector Convert floating-point to integer with Round towards -Infinity converts each element in a vector from floating-point to integer using the Round towards -Infinity rounding mode, and places the results in a second vector. The operand vector elements are floating-point numbers. The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCVTM{<q>}.<dt>.<dt2> <Od>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && 'HaveFP16Ext') || size IN {'00', '11'} then UNDEFINED;
rounding = FPDencodeRM(RM); unsigned = (op == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | Vd | 0 | 0 | 1 | 1 | op | Q | M | 0 | Vm |
| RM                                                                                                                                                                                                     |

64-bit SIMD vector (Q == 0)

VCVTM{<q>}.<dt>.<dt2> <Od>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && 'HaveFP16Ext') || size IN {'00', '11'} then UNDEFINED;
rounding = FPDencodeRM(RM); unsigned = (op == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler Symbols**

- `<q>`: See *Standard assembler syntax fields*.
- `<dt>`: Is the data type for the elements of the destination, encoded in “op:size”:

<table>
<thead>
<tr>
<th>op</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

- `<dt2>`: Is the data type for the elements of the source vector, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.`
- `<Qm>`: Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.`
- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>`: Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
  for e = 0 to elements-1
    Elem[D[d+r], e, esize] = FPToFixed(Elem[D[m+r], e, esize], 0, unsigned, StandardFPSCRValue(), rounding);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCVTM (floating-point)

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register. Depending on settings in the CPACR, NSACR, HCPR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 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 1 1 0 1 | D | 1 1 1 1 1 1 | Vd | 1 0 | != 00 | op | 1 | M | 0 | Vm |
| RM size |

Half-precision scalar (size == 01)
(Armv8.2)

VCVTM{q}.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTM{q}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTM{q}.<dt>.F64 <Sd>, <Sm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 1 1 1 1 0 1 | D | 1 1 1 1 1 1 | Vd | 1 0 | != 00 | op | 1 | M | 0 | Vm |
| RM size |
### Half-precision scalar (size == 01)  
(Armv8.2)

\[
\text{VCVTM}\{<q>\}.<\text{dt}>.F16 \ <\text{Sd}>, \ <\text{Sm}>
\]

### Single-precision scalar (size == 10)

\[
\text{VCVTM}\{<q>\}.<\text{dt}>.F32 \ <\text{Sd}>, \ <\text{Sm}>
\]

### Double-precision scalar (size == 11)

\[
\text{VCVTM}\{<q>\}.<\text{dt}>.F64 \ <\text{Sd}>, \ <\text{Dm}>
\]

If `InITBlock()` then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
  when '01'  esize = 16; m = UInt(Vm:M);
  when '10'  esize = 32; m = UInt(Vm:M);
  when '11'  esize = 64; m = UInt(M:Vm);

---

### CONstrained UNPREDIcTAbLe behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

### Assembler Symbols

- `<q>` See *Standard assembler syntax fields*.
- `<dt>` Is the data type for the elements of the destination, encoded in "op":

<table>
<thead>
<tr>
<th>op</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U32</td>
</tr>
<tr>
<td>1</td>
<td>S32</td>
</tr>
</tbody>
</table>

- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation

\[
\text{EncodingSpecificOperations(); CheckVFPEnabled(TRUE)};
\]

case esize of
  when 16
    \[\text{S}[d] = \text{FPToFixed}(\text{S}[m]<15:0>, 0, \text{unsigned}, \text{FPSCR}[], \text{rounding});\]
  when 32
    \[\text{S}[d] = \text{FPToFixed}(\text{S}[m], 0, \text{unsigned}, \text{FPSCR}[], \text{rounding});\]
  when 64
    \[\text{S}[d] = \text{FPToFixed}(\text{D}[m], 0, \text{unsigned}, \text{FPSCR}[], \text{rounding});\]

---

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCVTN (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest converts each element in a vector from floating-point to integer using the Round to Nearest rounding mode, and places the results in a second vector. The operand vector elements are floating-point numbers. The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCVTN{q}.<dt>.<dt2> <Od>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTN{q}.<dt>.<dt2> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' | | Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) | | size IN {'00', '11'} then UNDEFINED;
rounding = FPDWordRM(RM); unsigned = (op == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>1</td>
<td>1</td>
<td>Vd</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VCVTN{q}.<dt>.<dt2> <Od>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTN{q}.<dt>.<dt2> <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' | | Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) | | size IN {'00', '11'} then UNDEFINED;
rounding = FPDWordRM(RM); unsigned = (op == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:
The instruction is UNDEFINED.
The instruction executes as if it passes the Condition code check.
The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

qd Is the data type for the elements of the destination, encoded in "op:size":

<table>
<thead>
<tr>
<th>op</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

dt Is the data type for the elements of the source vector, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

qd Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as qd*2.

qm Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as qm*2.

dd Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

dm Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned, StandardFPSCRValue(), rounding);

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VCVTN (floating-point)

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in a second register. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & D & 1 & 1 & 1 & 1 & 0 & 1 & Vd & 1 & 0 & != & 0 & op & 1 & M & 0 & Vm \\
\end{array}
\]

RM size

Half-precision scalar (size == 01) (Armv8.2)

\[ \text{VCVTN}\{<q>\}.<dt>.F16 <Sd>, <Sm> \]

Single-precision scalar (size == 10)

\[ \text{VCVTN}\{<q>\}.<dt>.F32 <Sd>, <Sm> \]

Double-precision scalar (size == 11)

\[ \text{VCVTN}\{<q>\}.<dt>.F64 <Sd>, <Sm> \]

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
case size of
when '01' esize = 16; m = UInt(Vm:M);
when '10' esize = 32; m = UInt(Vm:M);
when '11' esize = 64; m = UInt(M:Vm);

T1

\[
\begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & D & 1 & 1 & 1 & 1 & 0 & 1 & Vd & 1 & 0 & != & 0 & op & 1 & M & 0 & Vm \\
\end{array}
\]

RM size
Half-precision scalar (size == 01)
(Armv8.2)

```
VCVTN<q>.<dt>F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVTN<q>.<dt>F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTN<q>.<dt>F64 <Sd>, <Dm>
```

if InIBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);

```c
#include
```
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);

CONstrained UNPREDICTABLE behavior

If InIBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- `<q>` See [Standard assembler syntax fields](#).
- `<dt>` Is the data type for the elements of the destination, encoded in "op":
  - **op** | **<dt>**
  - 0 | **U32**
  - 1 | **S32**

- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
      S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
  when 32
      S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
  when 64
      S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
```
VCVTP (Advanced SIMD)

Vector Convert floating-point to integer with Round towards +Infinity converts each element in a vector from floating-point to integer using the Round towards +Infinity rounding mode, and places the results in a second vector.
The operand vector elements are floating-point numbers.
The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.
Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VCVTP{q}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTP{q}.<dt>.<dt2> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | D | 1  | 1  | size | 1  | 1  | Vd | 0  | 0  | 1  | op | Q | M | 0  | Vm |
| RM |

64-bit SIMD vector (Q == 0)

VCVTP{q}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTP{q}.<dt>.<dt2> <Qd>, <Qm>

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRANGED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:
The instruction is UNDEFINED.
The instruction executes as if it passes the Condition code check.
The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

\[<q>\]
See Standard assembler syntax fields.

\[<dt>\]
Is the data type for the elements of the destination, encoded in "op:size":

\[
\begin{array}{cc}
\text{op} & \text{size} \\
0 & 01 \quad \text{S16} \\
0 & 10 \quad \text{S32} \\
1 & 01 \quad \text{U16} \\
1 & 10 \quad \text{U32} \\
\end{array}
\]

\[<dt2>\]
Is the data type for the elements of the source vector, encoded in "size":

\[
\begin{array}{c}
\text{size} \\
01 \\
10 \\
\end{array}
\]

\[<Qd>\]
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>\times 2\).

\[<Qm>\]
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as \(<Qm>\times 2\).

\[<Dd>\]
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\[<Dm>\]
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations();  CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
  for e = 0 to elements-1
    \[\text{Elem} [D[d+r],e,esize] = \text{FPToFixed} (\text{Elem} [D[m+r],e,esize], 0, \text{unsigned}, \text{StandardFPSCRValue} (), \text{rounding});\]
VCVTP (floating-point)

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in a second register. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 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 1 1 1 0 1 D 1 1 1 1 1 1 0 | Vd | 1 0 | != 00 | op | 1 | M | 0 | Vm |

RM size

Half-precision scalar (size == 01) (Armv8.2)

VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTP{<q>}.<dt>.F64 <Sd>, <Sm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd); case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------------------------|-----------------|-----------------|
| 1 1 1 1 1 1 1 1 0 1 D 1 1 1 1 1 1 0 | Vd | 1 0 | != 00 | op | 1 | M | 0 | Vm |

RM size
Half-precision scalar (size == 01)
(Armv8.2)

VCVTP<q>.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTP<q>.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTP<q>.<dt>.F64 <Sd>, <Sm>
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
CONSTRANDED UNPREDICTABLE behavior
If InITBlock(), then one of the following behaviors must occur:
  • The instruction is UNDEFINED.
  • The instruction executes as if it passes the Condition code check.
  • The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the destination, encoded in "op":

<table>
<thead>
<tr>
<th>op</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U32</td>
</tr>
<tr>
<td>1</td>
<td>S32</td>
</tr>
</tbody>
</table>
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
**VCVTR**

Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by the **FPSCR** and places the result in a second register.

**VCVT (between floating-point and fixed-point, floating-point)** describes conversions between floating-point and 16-bit integers.

Depending on settings in the **CPACR**, **NSACR**, **HCPTR**, and **FPEXC** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ( **A1** ) and T32 ( **T1** ).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 1 | 0 | x | Vd | 1 | 0 | size | 0 | 1 | M | 0 | Vm |
| cond | opc2 | op |

---

**VCVTR**

Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by the **FPSCR** and places the result in a second register.

**VCVT (between floating-point and fixed-point, floating-point)** describes conversions between floating-point and 16-bit integers.

Depending on settings in the **CPACR**, **NSACR**, **HCPTR**, and **FPEXC** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ( **A1** ) and T32 ( **T1** ).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 1 | 0 | x | Vd | 1 | 0 | size | 0 | 1 | M | 0 | Vm |
| cond | opc2 | op |
Half-precision scalar (opc2 == 100 && size == 01) (Armv8.2)

\[
\text{VCVTR}\{c\}\{q\}.U32.F16 \quad <Sd>, \quad <Sm>
\]

Half-precision scalar (opc2 == 101 && size == 01) (Armv8.2)

\[
\text{VCVTR}\{c\}\{q\}.S32.F16 \quad <Sd>, \quad <Sm>
\]

Single-precision scalar (opc2 == 100 && size == 10)

\[
\text{VCVTR}\{c\}\{q\}.U32.F32 \quad <Sd>, \quad <Sm>
\]

Single-precision scalar (opc2 == 101 && size == 10)

\[
\text{VCVTR}\{c\}\{q\}.S32.F32 \quad <Sd>, \quad <Sm>
\]

Double-precision scalar (opc2 == 100 && size == 11)

\[
\text{VCVTR}\{c\}\{q\}.U32.F64 \quad <Sd>, \quad <Dm>
\]

Double-precision scalar (opc2 == 101 && size == 11)

\[
\text{VCVTR}\{c\}\{q\}.S32.F64 \quad <Sd>, \quad <Dm>
\]

if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
  unsigned = (opc2<0> == '0');
  rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
d = UInt(Vd:D);
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
else
  unsigned = (op == '0');
  rounding = FPRoundingMode(FPSCR[]);
m = UInt(Vm:M);
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

\[
\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & D & 1 & 1 & 1 & 1 & 0 & x & Vd & 1 & 0 & \hline
\end{array}
\]

opc2 op Vm
Half-precision scalar (opc2 == 100 && size == 01)
(Armv8.2)

VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Half-precision scalar (opc2 == 101 && size == 01)
(Armv8.2)

VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar (opc2 == 100 && size == 10)

VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar (opc2 == 101 && size == 10)

VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar (opc2 == 100 && size == 11)

VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Sm>

Double-precision scalar (opc2 == 101 && size == 11)

VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Sm>

if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
  unsigned = (opc2<0> == '0');
  rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR());
  d = UInit(Vd:D);
  case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
  else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR());
    m = UInt(Vm:M);
    case size of
      when '01' esize = 16; d = UInt(Vd:D);
      when '10' esize = 32; d = UInt(Vd:D);
      when '11' esize = 64; d = UInt(D:Vd);
  end

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Floating-point data-processing for the T32 instruction set, or Floating-point data-processing for the A32 instruction set.

Assembler Symbols

<
See Standard assembler syntax fields.

> See Standard assembler syntax fields.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
  if to_integer then
    case esize of
      when 16  
        \$d = FTPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
      when 32  
        \$d = FTPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
      when 64  
        \$d = FTPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
  else
    case esize of
      when 16
        bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
        \$d = Zeros(16):fp16;
      when 32
        \$d = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
      when 64
        D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
```
VCVTT

Convert to or from a half-precision value in the top half of a single-precision register does one of the following:

- Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|------------------|------------------|
| != 1111 | 1 1 1 0 | D | 1 1 0 | 0 0 | op | Vd | 1 0 | 1 | sz | 1 1 | M | 0 | Vm |
| cond   | T      |

Half-precision to single-precision (op == 0 & sz == 0)

VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Half-precision to double-precision (op == 0 & sz == 1)

VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Single-precision to half-precision (op == 1 & sz == 0)

VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Double-precision to half-precision (op == 1 & sz == 1)

VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>

uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
if uses_double then
  if convert_from_half then
    d = UInt(D:Vd); m = UInt(Vm:M);
  else
    d = UInt(Vd:D); m = UInt(M:Vm);
else
  d = UInt(Vd:D); m = UInt(Vm:M);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|------------------|------------------|
| 1 1 1 0 | 1 1 1 0 | D | 1 1 0 | 0 0 | op | Vd | 1 0 | 1 | sz | 1 1 | M | 0 | Vm |
| T      |
Half-precision to single-precision (\(op == 0 \&\& sz == 0\))

\[
\text{VCVTT\{<c>\}{<q>}.F32.F16 <Sd>, <Sm>}
\]

Half-precision to double-precision (\(op == 0 \&\& sz == 1\))

\[
\text{VCVTT\{<c>\}{<q>}.F64.F16 <Dd>, <Sm>}
\]

Single-precision to half-precision (\(op == 1 \&\& sz == 0\))

\[
\text{VCVTT\{<c>\}{<q>}.F16.F32 <Sd>, <Sm>}
\]

Double-precision to half-precision (\(op == 1 \&\& sz == 1\))

\[
\text{VCVTT\{<c>\}{<q>}.F16.F64 <Sd>, <Dm>}
\]

\[
\text{uses_double = (sz == '1'); convert_from_half = (op == '0');}
\text{lowbit = (if T == '1' then 16 else 0);}
\]

\[
\text{if uses_double then}
\text{if convert_from_half then}
\text{\quad d = \text{UInt}(Vd:D); m = \text{UInt}(M:Vm);} \\
\text{else}
\text{\quad d = \text{UInt}(Vd:D); m = \text{UInt}(M:Vm);} \\
\text{else}
\text{\quad d = \text{UInt}(Vd:D); m = \text{UInt}(M:Vm);} \\
\]

Assembler Symbols

- \(<c>\): See \textit{Standard assembler syntax fields}.
- \(<q>\): See \textit{Standard assembler syntax fields}.
- \(<Sd>\): Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- \(<Dm>\): Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- \(<Dd>\): Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- \(<Sm>\): Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

\[
\text{if ConditionPassed() then}
\text{\quad EncodingSpecificOperations(); \text{CheckVFPEnabled}(TRUE);} \\
\]

\[
\text{\quad bits(16) hp;}
\text{\quad if convert_from_half then}
\text{\quad \quad hp = S[m]<lowbit+15:lowbit>;} \\
\text{\quad \quad if uses_double then}
\text{\quad \quad \quad D[d] = \text{FPConvert}(hp, FPSCR[]);} \\
\text{\quad \quad \quad else}
\text{\quad \quad \quad S[d] = \text{FPConvert}(hp, FPSCR[]);} \\
\text{\quad \quad else}
\text{\quad \quad \quad if uses_double then}
\text{\quad \quad \quad \quad hp = \text{FPConvert}(D[m], FPSCR[]);} \\
\text{\quad \quad \quad \quad else}
\text{\quad \quad \quad \quad hp = \text{FPConvert}(S[m], FPSCR[]);} \\
\text{\quad \quad \quad S[d]<lowbit+15:lowbit> = hp;}
\]
VCVTT (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result in the top half of a single-precision register, preserving the bottom 16 bits of the register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the FPSCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the FPSCR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPSCR.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1 (Armv8.6)

\[
\begin{array}{ccccccccccccccccccccccc}
\hline
1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & Vd & 1 & 0 & 0 & 1 & 1 & 1 & M & 0 & Vm \\
\end{array}
\]

cond

if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);

T1 (Armv8.6)

\[
\begin{array}{ccccccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & Vd & 1 & 0 & 0 & 1 & 1 & 1 & M & 0 & Vm \\
\end{array}
\]

if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<Sd>\) Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
\(<Sm>\) Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    \[ S[d]<31:16> = FPConvertBE(S[m], FPSCR[]) \]
DIV

Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.
Depending on settings in the `CPACR`, `NSACR`, `HCPTR`, and `FPEXC` registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| != 1111 | 1 1 1 0 | 1 | D | 0 | 0 | Vn | Vd | 1 | 0 | size | N | 0 | M | 0 | Vm |
```

cond

Half-precision scalar (size == 01)
(ARMv8.2)

```
VDIV{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONstrained UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | D | 0 | 0 | Vn | Vd | 1 | 0 | size | N | 0 | M | 0 | Vm |
```
Half-precision scalar (size == 01)  
(Armv8.2)

VDIV{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>

Single-precision scalar (size == 10)

VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>

Double-precision scalar (size == 11)

VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>

if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRUANED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<e> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPOEMDEnabled(TRUE);
case esize of
    when 16
        S[d] = Zerps(16) : FPDiv(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
    when 32
        S[d] = FPDiv(S[n], S[m], FPSCR[]);
    when 64
        D[d] = FPDiv(D[n], D[m], FPSCR[]);

VDOT (vector)

BFloat16 floating-point (BF16) dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BF16 values in the first source vector. The instruction does not update the FPSCR exception status.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1
(Armv8.6)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 0  | 0  | 0  | D  | 0  | 0  | Vn | Vd | 1  | 1  | 0  | 1  | N  | Q  | M  | 0  | Vm |

64-bit SIMD vector (Q == 0)

VDOT({<q>}.BF16 <Dd>, <Dn>, <Dm>)

128-bit SIMD vector (Q == 1)

VDOT({<q>}.BF16 <Qd>, <Qn>, <Qm>)

if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;

T1
(Armv8.6)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | D  | 0  | 0  | Vn | Vd | 1  | 1  | 0  | 1  | N  | Q  | M  | 0  | Vm |

64-bit SIMD vector (Q == 0)

VDOT({<q>}.BF16 <Dd>, <Dn>, <Dm>)

128-bit SIMD vector (Q == 1)

VDOT({<q>}.BF16 <Qd>, <Qn>, <Qm>)

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;

Assembler Symbols

<q> See Standard assembler syntax fields.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```c
bits(64) operand1;
bits(64) operand2;
bits(64) result;

CheckAdvSIMDEnabled();

for r = 0 to regs-1
    operand1 = Din[n+r];
    operand2 = Din[m+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(16) elt1_a = Elem(operand1, 2 * e + 0, 16);
        bits(16) elt1_b = Elem(operand1, 2 * e + 1, 16);
        bits(16) elt2_a = Elem(operand2, 2 * e + 0, 16);
        bits(16) elt2_b = Elem(operand2, 2 * e + 1, 16);
        bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
        Elem[result, e, 32] = BFAdd(Elem[result, e, 32], sum);
    D[d+r] = result;
```
VDOT (by element)

BFloat16 floating-point indexed dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the indexed pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BFloat16 values in the first source vector. The instruction does not update the FPSCR exception status.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1
(Armv8.6)

```
31 30 29 28 27 26 25 24 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 1 0 | 0 | D | 0 | 0 | Vn | Vd | 1 1 0 | 1 | N | Q | M | 0 | Vm
```

64-bit SIMD vector (Q == 0)

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector (Q == 1)

VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;

T1
(Armv8.6)

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 0 | 0 | D | 0 | 0 | Vn | Vd | 1 1 0 | 1 | N | Q | M | 0 | Vm
```

64-bit SIMD vector (Q == 0)

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector (Q == 1)

VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;

Assembler Symbols

$q$ See Standard assembler syntax fields.
$qd$ Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \(<Qn>^2\).

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

Is the element index in the range 0 to 1, encoded in the "M" field.

**Operation**

```c
bits(64) operand1;
bits(64) operand2;
bits(64) result;

CheckAdvSIMDEnabled();

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        bits(16) elt2_a = Elem[operand2, 2 * i + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * i + 1, 16];
        bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
        Elem[result, e, 32] = BFAdd(Elem[result, e, 32], sum);
    D[d+r] = result;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**VDUP (general-purpose register)**

Duplicate general-purpose register to vector duplicates an element from a general-purpose register into every element of the destination vector.

The destination vector elements can be 8-bit, 16-bit, or 32-bit fields. The source element is the least significant 8, 16, or 32 bits of the general-purpose register. There is no distinction between data types.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|
| != 1111 | 1 | 1 | 1 | 0 | 1 | B | Q | 0 | Vd | | Rt | 1 | 0 | 1 | 1 | D | 0 | E | 1 | (0)(0)(0)(0)(0) |

**cond**

#### A1

**VDUP**

- `<c>`{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)
- `<c>`{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)

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

\[
d = \text{UInt}(D:Vd); \quad t = \text{UInt}(Rt); \quad \text{regs} = \text{if Q == '0' then 1 else 2};
\]

**case** B:E of

- when '00' esize = 32; elements = 2;
- when '01' esize = 16; elements = 4;
- when '10' esize = 8; elements = 8;
- when '11' UNDEFINED;

if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | B | Q | 0 | Vd | | Rt | 1 | 0 | 1 | 1 | D | 0 | E | 1 | (0)(0)(0)(0)(0) |

#### T1

**VDUP**

- `<c>`{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)
- `<c>`{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)

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

\[
d = \text{UInt}(D:Vd); \quad t = \text{UInt}(Rt); \quad \text{regs} = \text{if Q == '0' then 1 else 2};
\]

**case** B:E of

- when '00' esize = 32; elements = 2;
- when '01' esize = 16; elements = 4;
- when '10' esize = 8; elements = 8;
- when '11' UNDEFINED;

if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<c>` See *Standard assembler syntax fields*. Arm strongly recommends that any VDUP instruction is unconditional, see *Conditional execution*.
- `<q>` See *Standard assembler syntax fields*. 

VDUP (general-purpose register)
The data size for the elements of the destination vector. It must be one of:

- **8**
  Encoded as \([b, e] = 0b10\).
- **16**
  Encoded as \([b, e] = 0b01\).
- **32**
  Encoded as \([b, e] = 0b00\).

The destination vector for a quadword operation.

The destination vector for a doubleword operation.

The Arm source register.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    scalar = R[t]<esize-1:0>;
    for r = 0 to regs-1
        for e = 0 to elements-1
            \( \text{Elem}[d+r],e,\text{esize}] = \text{scalar} ;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Duplicate vector element to vector duplicates a single element of a vector into every element of the destination vector. The scalar, and the destination vector elements, can be any one of 8-bit, 16-bit, or 32-bit fields. There is no distinction between data types.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|----------------|----------------|----------------|
| D 1 1 1 0 0 1 1 1 imm4 Vd | 1 1 0 0 0 Q M 0 Vm |

(Q == 0)

VDUP{<c>{<q>}.<size> <Qd>, <Dm[x]>

(Q == 1)

VDUP{<c>{<q>}.<size> <Qd>, <Dm[x]>

if imm4 == 'x000' then UNDEFINED;
if Q == '1' & & Vd<0> == '1' then UNDEFINED;
case imm4 of
  when 'xxx1' esize = 8; elements = 8; index = UInt(imm4<3:1>);
  when 'xx10' esize = 16; elements = 4; index = UInt(imm4<3:2>);
  when 'x100' esize = 32; elements = 2; index = UInt(imm4<3>);
  d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------|----------------|----------------|----------------|
| 1 1 1 1 1 1 1 1 1 1 imm4 Vd | 1 1 0 0 0 Q M 0 Vm |

(Q == 0)

VDUP{<c>{<q>}.<size> <Qd>, <Dm[x]>

(Q == 1)

VDUP{<c>{<q>}.<size> <Qd>, <Dm[x]>

if imm4 == 'x000' then UNDEFINED;
if Q == '1' & & Vd<0> == '1' then UNDEFINED;
case imm4 of
  when 'xxx1' esize = 8; elements = 8; index = UInt(imm4<3:1>);
  when 'xx10' esize = 16; elements = 4; index = UInt(imm4<3:2>);
  when 'x100' esize = 32; elements = 2; index = UInt(imm4<3>);
  d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.
<size> The data size. It must be one of:

- **8**
  - Encoded as imm4<0> = '1'. imm4<3:1> encodes the index[x] of the scalar.

- **16**
  - Encoded as imm4<1:0> = '10'. imm4<3:2> encodes the index [x] of the scalar.

- **32**
  - Encoded as imm4<2:0> = '100'. imm4<3> encodes the index [x] of the scalar.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm[x]> The scalar. For details of how [x] is encoded, see the description of <size>.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    scalar = Elem[D[m],index,esize];
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

**Operational information**

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

Vector Bitwise Exclusive OR performs a bitwise Exclusive OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

![A1 encoding](image)

64-bit SIMD vector (Q == 0)

VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

128-bit SIMD vector (Q == 1)

VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**T1**

![T1 encoding](image)

64-bit SIMD vector (Q == 0)

VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**Assembler Symbols**

- `<c>` For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
  For encoding T1: see Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.
- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- `<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.
- `<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
Operation

if `ConditionPassed()` then
    EncodingSpecificOperations();  `CheckAdvSIMDEnabled()`;
    for r = 0 to regs-1
        \( \text{D}[d+r] = \text{D}[n+r] \text{ EOR D}[m+r]; \)

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

The elements of the vectors are treated as being 8-bit fields. There is no distinction between data types.

The following figure shows an example of the operation of VEXT doubleword operation for imm = 3.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instruction VEXT (multibyte elements).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | D  | 1  | 1  | Vn | Vd | imm4 | N  | Q  | M  | 0  | Vm |

64-bit SIMD vector (Q == 0)

VEXT{<c>}{<q>}.8 {<Dd>,} {<Dn>,} {<Qd>,} {<Qn>,} #<imm>

128-bit SIMD vector (Q == 1)

VEXT{<c>}{<q>}.8 {<Dd>,} {<Dn>,} {<Qd>,} {<Qn>,} #<imm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quardword_operation = (Q == '1'); position = 8 * UInt(imm4);
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | D  | 1  | 1  | Vn | Vd | imm4 | N  | Q  | M  | 0  | Vm |

64-bit SIMD vector (Q == 0)

VEXT{<c>}{<q>}.8 {<Dd>,} {<Dn>,} {<Qd>,} {<Qn>,} #<imm>

128-bit SIMD vector (Q == 1)

VEXT{<c>}{<q>}.8 {<Dd>,} {<Dn>,} {<Qd>,} {<Qn>,} #<imm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quardword_operation = (Q == '1'); position = 8 * UInt(imm4);
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
Assembler Symbols

For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

See *Standard assembler syntax fields*.

**<Qd>** Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

**<Qn>** Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

**<Qm>** Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**<Dn>** Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

**<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**<imm>** For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 7, encoded in the "imm4" field.

For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 15, encoded in the "imm4" field.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    if quadword_operation then
        Q[d>>1] = (Q[m>>1]:Q[n>>1])<position+127:position>;
    else
        Q[d] = (Q[m]:Q[n])<position+63:position>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

This is a pseudo-instruction of **VEXT (byte elements)**. This means:

- The encodings in this description are named to match the encodings of **VEXT (byte elements)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **VEXT (byte elements)** gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 1 1 0 0 1 0 1 D 1 1 | Vn | Vd | imm4 | N | Q | M | 0 | Vm |

**64-bit SIMD vector (Q == 0)**

VEXT{<c>{<q>.<size> {<Dd>,} <Dn>, <Dm>, #<imm} is equivalent to

VEXT{<c>{<q>.8 {<Dd>,} <Dn>, <Dm>, #<imm*(size/8)}

**128-bit SIMD vector (Q == 1)**

VEXT{<c>{<q}.<size> {<Qd>,} <Qn>, <Qm>, #<imm} is equivalent to

VEXT{<c>{<q}.8 {<Qd>,} <Qn>, <Qm>, #<imm*(size/8)}

### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 0 0 1 1 1 1 1 D 1 1 | Vn | Vd | imm4 | N | Q | M | 0 | Vm |

**64-bit SIMD vector (Q == 0)**

VEXT{<c>{<q}.<size> {<Dd>,} <Dn>, <Dm>, #<imm} is equivalent to

VEXT{<c>{<q}.8 {<Dd>,} <Dn>, <Dm>, #<imm*(size/8)}

**128-bit SIMD vector (Q == 1)**

VEXT{<c>{<q}.<size> {<Qd>,} <Qn>, <Qm>, #<imm} is equivalent to

VEXT{<c>{<q}.8 {<Qd>,} <Qn>, <Qm>, #<imm*(size/8)}

### Assembler Symbols

<
c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<
q> See *Standard assembler syntax fields*. 

---

VEXT (multibyte elements)
For the 64-bit SIMD vector variant: is the size of the operation, and can be one of 16 or 32.

For the 128-bit SIMD vector variant: is the size of the operation, and can be one of 16, 32 or 64.

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>  Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<imm> For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (128/<size>)-1.

For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (64/<size>)-1.

**Operation**

The description of **VEXT (byte elements)** gives the operational pseudocode for this instruction.
VFMA

Vector Fused Multiply Accumulate multiplies corresponding elements of two vectors, and accumulates the results into the elements of the destination vector. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPSCR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

\[
\begin{array}{cccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & D & 0 & sz & Vn & Vd & 1 & 1 & 0 & N & Q & M & 1 & Vm & op
\end{array}
\]

64-bit SIMD vector (Q == 0)

VFMA{<c>}{<q>}{<dt> & <Dd>, <Dn>, <Dm>}

128-bit SIMD vector (Q == 1)

VFMA{<c>}{<q>}{<dt> & <Qd>, <Qn>, <Qm>}

if Q == '1' & & (Vd<0> == '1' | | Vn<0> == '1' | | Vm<0> == '1') then UNDEFINED;
if sz == '1' & & HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1');
case sz of
when '0' esize = 32; elements = 2;
when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;

A2

\[
\begin{array}{cccccccccccccccccc}
!= 1111 & 1 & 1 & 1 & 0 & 1 & D & 1 & 0 & Vn & Vd & 1 & 0 & size & N & 0 & M & 0 & Vm & cond & op
\end{array}
\]

Half-precision scalar (size == 01)

(Armv8.2)

VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' | | FPSCR.Stride != '00' then UNDEFINED;
if size == '00' | | (size == '01' & & !HaveFP16Ext()) then UNDEFINED;
if size == '01' & & cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
CONSTRANGED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');

CONSTRANGED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
</tbody>
</table>
Half-precision scalar (size == 01)
(Armv8.2)
VFMA{<c}>{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)
VFMA{<c}>{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)
VFMA{<c}>{<q>}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRANGED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:
  • The instruction is UNDEFINED.
  • The instruction executes as if it passes the Condition code check.
  • The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>  For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
     For encoding A2, T1 and T2: see Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
if `ConditionPassed()` then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                bits(esize) op1 = `Elem[D[n+r],e,esize];
                if op1_neg then op1 = `FPNeg(op1);
                `Elem[D[d+r],e,esize] = `FPMulAdd(Elem[D[d+r],e,esize],
                                     op1, `Elem[D[m+r],e,esize], `StandardFPSCRValue());
    else // VFP instruction
        case esize of
            when 16
                op16 = if op1_neg then `FPNeg(S[n]<15:0>) else S[n]<15:0>;
                `S[d] = `Zeros(16): `FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR[]);
            when 32
                op32 = if op1_neg then `FPNeg(S[n]) else S[n];
                `S[d] = `FPMulAdd(S[d], op32, S[m], FPSCR[]);
            when 64
                op64 = if op1_neg then `FPNeg(D[n]) else D[n];
                `D[d] = `FPMulAdd(D[d], op64, D[m], FPSCR[]);

VFMAB, VFMAT (BFLOAT16, vector)

The BFloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from BFloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1 (Armv8.6)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | 0  | D  | 1  | 1  | Vn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1  | 0  | 0  | 0  | N  | Q  | M  | 1  | Vm |
```

A1

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
integer sel = UInt(Q);

T1 (Armv8.6)

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | D  | 1  | 1  | Vn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1  | 0  | 0  | 0  | N  | Q  | M  | 1  | Vm |
```

T1

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
integer sel = UInt(Q);

Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in "Q":

<table>
<thead>
<tr>
<th></th>
<th>&lt;bt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
</tbody>
</table>

$q$ See Standard assembler syntax fields.

<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation

CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) operand3 = Q[d>>1];
bits(128) result;

for e = 0 to elements-1
  bits(32) element1 = Elem[operand1, 2 * e + sel, 16] : Zeros(16);
  bits(32) element2 = Elem[operand2, 2 * e + sel, 16] : Zeros(16);
  bits(32) addend = Elem[operand3, e, 32];
  Elem[result, e, 32] = FPMulAdd(addend, element1, element2,
      StandardFPSCRValue());

Q[d>>1] = result;
VFMAB, VFMAT (BFloat16, by scalar)

The BFloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and an indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1
(Armv8.6)

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & D & 1 & 1 & Vn & Vd & 1 & 0 & 0 & 0 & N & Q & M & 1 & Vm
\end{array}
\]

A1
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == ‘1’ || Vn<0> == ‘1’ then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
inT<1>vent i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);

T1
(Armv8.6)

\[
\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & D & 1 & 1 & Vn & Vd & 1 & 0 & 0 & 0 & N & Q & M & 1 & Vm
\end{array}
\]

T1
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == ‘1’ || Vn<0> == ‘1’ then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
inT<1>vent i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);

Assembler Symbols

\(<\text{bt}>\) Is the bottom or top element specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;\text{bt}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
</tr>
</tbody>
</table>

\(<\text{q}>\) See Standard assembler syntax fields.

\(<\text{Qd}>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

\(<\text{Qn}>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
Operation

CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(64) operand2 = D[m];
bits(128) operand3 = Q[d>>1];
bits(128) result;

bits(32) element2 = Elem(operand2, i, 16) : Zeros(16);
for e = 0 to elements-1
    bits(32) element1 = Elem(operand1, 2 * e + sel, 16) : Zeros(16);
    bits(32) addend = Elem(operand3, e, 32);
    Elem(result, e, 32) = FPMulAdd(addend, element1, element2, StandardFPSCRValue());
Q[d>>1] = result;
VFMAL (vector)

Vector Floating-point Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding 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.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1
(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  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | D | 1  | 0  | Vn | Vd | 1  | 0  | 0  | 0  | N | Q | M | 1  | Vm |

64-bit SIMD vector (Q == 0)

VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>

128-bit SIMD vector (Q == 1)

VFMAL{<q>}.F16 <Dd>, <Dn>, <Dm>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' & Vd<8> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if 0 == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';

T1
(Armv8.2)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | D | 1  | 0  | Vn | Vd | 1  | 0  | 0  | 0  | N | Q | M | 1  | Vm |
64-bit SIMD vector (Q == 0)

\[ \text{VFMAL}\{<q>\}.F16 \ <Dd>, \ <Sn>, \ <Sm> \]

128-bit SIMD vector (Q == 1)

\[ \text{VFMAL}\{<q>\}.F16 \ <Qd>, \ <Dn>, \ <Dm> \]

if \( \text{InITBlock}() \) then UNPREDICTABLE;
if \( \text{HaveFP16MulNoRoundingToFP32Ext}() \) then UNDEFINED;
if \( Q == '1' && Vd<0> == '1' \) then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if \( Q == '1' \) then UInt(N:Vn) else UInt(Vn:N);
integer m = if \( Q == '1' \) then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if \( Q=='1' \) then 2 else 1;
integer datasize = if \( Q=='1' \) then 64 else 32;
boolean sub_op = S=='1';

**Assembler Symbols**

\(<q>\)\ See *Standard assembler syntax fields*.

\(<Qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>*2.\n
\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Sn>\) Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

\(<Sm>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

**Operation**

\( \text{CheckAdvSIMDEnabled}() \);

bits(datasize) operand1;
bits(datasize) operand2;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if \( Q=='0' \) then
  operand1 = \( S[n]<\text{datasize-1:0}>; \)
  operand2 = \( S[m]<\text{datasize-1:0}>; \)
else
  operand1 = \( D[n]<\text{datasize-1:0}>; \)
  operand2 = \( D[m]<\text{datasize-1:0}>; \)
for \( r = 0 \) to \( \text{regs}-1 \)
  operand3 = \( D[d+r]; \)
for \( e = 0 \) to \( 1 \)
  element1 = Elem(operand1, \( 2*r+e, \text{esize} \text{ DIV } 2 \));
  element2 = Elem(operand2, \( 2*r+e, \text{esize} \text{ DIV } 2 \));
  if \( \text{sub_op} \) then element1 = \( \text{FPNeg}(element1); \)
  Elem(result, e, esize) = FPMulAddH(Elem(operand3, e, esize), element1, element2, \text{StandardFPSCRVal});
  \( D[d+r] = \text{result}; \)
VFMAL (by scalar)

Vector Floating-point Multiply-Add Long to accumulator (by scalar). 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.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

**ID ISAR6.FHM** indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1 (Armv8.2)

![Instruction set encoding](image)

64-bit SIMD vector (Q == 0)

\[
\text{VFMAL}{<q>}.F16, <Dd>, <Sn>, <Sm>[<index>]
\]

128-bit SIMD vector (Q == 1)

\[
\text{VFMAL}{<q>}.F16, <Dd>, <Dn>, <Dm>[<index>]
\]

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<8> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datsize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';

### T1 (Armv8.2)

![Instruction set encoding](image)
64-bit SIMD vector \((Q == 0)\)

\[
\text{VFMAL\{<q>\}.F16} \ <\text{Dd}, \ <\text{Sn}, \ <\text{Sm}[<\text{index}>]}
\]

128-bit SIMD vector \((Q == 1)\)

\[
\text{VFMAL\{<q>\}.F16} \ <\text{Qd}, \ <\text{Dn}, \ <\text{Dm}[<\text{index}>]}
\]

if \text{InitBlock()} then UNPREDICTABLE;
if !\text{HaveFP16MulNoRoundingToFP32Ext()} then UNDEFINED;
if \(Q == '1' \&\& \text{Vd}<0> == '1'\) then UNDEFINED;

\[
\begin{align*}
\text{integer} &\ d = \text{UInt}(D:Vd); \\
\text{integer} &\ n = \text{if} \ Q == '1' \text{ then } \text{UInt}(N:Vn) \text{ else } \text{UInt}(Vn:N); \\
\text{integer} &\ m = \text{if} \ Q == '1' \text{ then } \text{UInt}(Vm<2:0>) \text{ else } \text{UInt}(Vm<2:0>:M); \\
\text{integer} &\ \text{index} = \text{if} \ Q == '1' \text{ then } \text{UInt}(M:Vm<3>) \text{ else } \text{UInt}(Vm<3>); \\
\text{integer} &\ \text{esize} = 32; \\
\text{integer} &\ \text{regs} = \text{if } Q == '1' \text{ then } 2 \text{ else } 1; \\
\text{integer} &\ \text{datasize} = \text{if } Q == '1' \text{ then } 64 \text{ else } 32; \\
\text{boolean} &\ \text{sub_op} = S == '1';
\end{align*}
\]

**Assembler Symbols**

- \(<q>\): See *Standard assembler syntax fields*.
- \(<Qd>\): Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- \(<Dn>\): Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- \(<Dm>\): Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.
- \(<Dd>\): Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- \(<Sn>\): Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
- \(<Sm>\): Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.
- \(<\text{index}>\): For the 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field.
  
  For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

**Operation**

\[
\begin{align*}
\text{CheckAdvSIMDEnabled}(); \\
\text{bits(datasize)} &\ \text{operand1} ; \\
\text{bits(datasize)} &\ \text{operand2} ; \\
\text{bits(64)} &\ \text{operand3}; \\
\text{bits(64)} &\ \text{result}; \\
\text{bits(esize DIV 2)} &\ \text{element1}; \\
\text{bits(esize DIV 2)} &\ \text{element2};
\end{align*}
\]

if \(Q == '0'\) then
\[
\begin{align*}
\text{operand1} &= S[n]<\text{datasize}-1:0>; \\
\text{operand2} &= S[m]<\text{datasize}-1:0>;
\end{align*}
\]

else
\[
\begin{align*}
\text{operand1} &= D[n]<\text{datasize}-1:0>; \\
\text{operand2} &= D[m]<\text{datasize}-1:0>;
\end{align*}
\]

\[
\begin{align*}
\text{element2} &= \text{Elem}[\text{operand2}, \ \text{index}, \ \text{esize DIV 2}];
\end{align*}
\]

for \(r = 0\) to \(\text{regs}-1\)
\[
\begin{align*}
\text{operand3} &= D[d+r]; \\
\text{for } e &= 0 \text{ to } 1 \\
\text{element1} &= \text{Elem}[\text{operand1}, \ 2*r+e, \ \text{esize DIV 2}]; \\
\text{if } \text{sub_op} \text{ then } \text{element1} &= \text{FPNeg}(\text{element1});
\end{align*}
\]

\[
\begin{align*}
\text{D[d+r]} &= \text{result};
\end{align*}
\]
VFMS

Vector Fused Multiply Subtract negates the elements of one vector and multiplies them with the corresponding elements of another vector, adds the products to the corresponding elements of the destination vector, and places the results in the destination vector. The instruction does not round the result of the multiply before the addition. Depending on settings in the CPACR, NSACR, HCPTTR, and FPSEX registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

64-bit SIMD vector (Q == 0)

VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

if Q == '1' & & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & & !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1');
case sz of
   when '0' esize = 32; elements = 2;
   when '1' esize = 16; elements = 4;
   d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
   regs = if Q == '0' then 1 else 2;

A2

Half-precision scalar (size == 01)
(Armv8.2)

VFMS{<c>}{<q>}.F16 <Dd>, <Dn>, <Dm>

Single-precision scalar (size == 10)

VFMS{<c>}{<q>}.F32 <Dd>, <Dn>, <Dm>

Double-precision scalar (size == 11)

VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' & & !HaveFP16Ext()) then UNDEFINED;
if size == '01' & & cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
   when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
   when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
   when '11' esize = 64; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
**CONSTRAINED UNPREDICTABLE behavior**

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | D | 1 | sz | Vn | Vd | 1 | 1 | 0 | N | Q | M | 1 | Vm |
| op |

64-bit SIMD vector (Q == 0)

`VFMS{<c>}{<q>}{<dt>}{<Dd>}, <Dn>, <Dm>`

128-bit SIMD vector (Q == 1)

`VFMS{<c>}{<q>}{<dt>}{<Qd>}, <Qn>, <Qm>`

if `Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1')` then UNDEFINED;
if `sz == '1' && !HaveFP16Ext()` then UNDEFINED;
if `sz == '1' && InITBlock()` then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;

**CONSTRAINED UNPREDICTABLE behavior**

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T2**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | D | 1 | 0 | Vn | Vd | 1 | 0 | size | N | 1 | M | 0 | Vm |
| op |
Half-precision scalar (size == 01)
(Armv8.2)

VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
ENDCASE;

CONSTRUAN UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;c&gt;</th>
<th>For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For encoding A2, T1 and T2: see Standard assembler syntax fields.</td>
</tr>
<tr>
<td>&lt;q&gt;</td>
<td>See Standard assembler syntax fields.</td>
</tr>
</tbody>
</table>
| <dt>| Is the data type for the elements of the vectors, encoded in “sz”:
|     | sz | <dt> |
|     | 0  | F32  |
|     | 1  | F16  |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.           |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.           |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.           |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.            |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.            |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.           |
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDorVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                bits(esize) op1 = Elem[D[n+r],e,esize];
                if op1_neg then op1 = FPNeg(op1);
                Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                                op1, Elem[D[m+r],e,esize], StandardFPSCRValue());

    else // VFP instruction
        case esize of
            when 16
                op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
                S[d] = Zeros(16) : FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR[]);
            when 32
                op32 = if op1_neg then FPNeg(S[n]) else S[n];
                S[d] = FPMulAdd(S[d], op32, S[m], FPSCR[]);
            when 64
                op64 = if op1_neg then FPNeg(D[n]) else D[n];
                D[d] = FPMulAdd(D[d], op64, D[m], FPSCR[]);

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VFMSL (vector)

Vector Floating-point 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.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it. The ID ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1 (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  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | D | 1  | 0  | Vn |    | Vd |    | 1  | 0  | 0  | 0  | 1  | N  | Q  | M  | 1  | Vm |

64-bit SIMD vector (Q == 0)

VFMSL{q}.F16 <Dd>, <Sn>, <Sm>

128-bit SIMD vector (Q == 1)

VFMSL{q}.F16 <Dd>, <Dn>, <Dm>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<8> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if O == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';

T1 (Armv8.2)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | D | 1  | 0  | Vn |    | Vd |    | 1  | 0  | 0  | 0  | 1  | N  | Q  | M  | 1  | Vm |
64-bit SIMD vector \((Q == 0)\)

**VFMSL**\(<q>\).F16 \(<Dd>\), \(<Sn>\), \(<Sm>\)

128-bit SIMD vector \((Q == 1)\)

**VFMSL**\(<q>\).F16 \(<Qd>\), \(<Dn>\), \(<Dm>\)

if \(\text{InITBlock}()\) then \text{UNPREDICTABLE};
if \(\text{HaveFP16MulNoRoundingToFP32Ext}()\) then \text{UNDEFINED};
if \(Q == '1' \&\& Vd<0> == '1'\) then \text{UNDEFINED};

integer \(d = \text{UInt}(D:Vd); \)
integer \(n = \) if \(Q == '1'\) then \(\text{UInt}(N:Vn)\) else \(\text{UInt}(Vn:N)\);
integer \(m = \) if \(Q == '1'\) then \(\text{UInt}(M:Vm)\) else \(\text{UInt}(Vm:M)\);
integer \(\text{esize} = 32;\)
integer \(\text{regs} = \) if \(Q=='1'\) then 2 else 1;
integer \(\text{datasize} = \) if \(Q=='1'\) then 64 else 32;
boolean \(\text{sub_op} = S=='1';\)

**Assembler Symbols**

\(<q>\) See **Standard assembler syntax fields**.

\(<Qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>*2.

\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

\(<Qd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Sn>\) Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

\(<Sm>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

**Operation**

\(\text{CheckAdvSIMDEnabled}();\)

bits(datasize) operand1;
bits(datasize) operand2;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if \(Q=='0'\) then

\(\text{operand1} = S[n]<\text{datasize}-1:0>;\)
\(\text{operand2} = S[m]<\text{datasize}-1:0>;\)
else

\(\text{operand1} = D[n]<\text{datasize}-1:0>;\)
\(\text{operand2} = D[m]<\text{datasize}-1:0>;\)

for \(r = 0\) to \(\text{regs}-1\)

\(\text{operand3} = D[d+r];\)

for \(e = 0\) to \(1\)

\(\text{element1} = \text{Elem}[\text{operand1}, 2*r+e, \text{esize} \text{ DIV } 2];\)
\(\text{element2} = \text{Elem}[\text{operand2}, 2*r+e, \text{esize} \text{ DIV } 2];\)

if \(\text{sub}_{\text{op}}\) then \(\text{element1} = \text{FPNeg}(\text{element1});\)

\(\text{Elem}[\text{result}, e, \text{esize}] = \text{FPMulAddH}[\text{Elem}[\text{operand3}, e, \text{esize}], \text{element1}, \text{element2}, \text{StandardFPSCRVal}];\)
\(D[d+r] = \text{result};\)
VFMSL (by scalar)

Vector Floating-point Multiply-Subtract Long from accumulator (by scalar). 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.

Depending on settings in the CPACR, NSACR, HCPR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it. ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1 (Armv8.2)

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

64-bit SIMD vector (Q == 0)

VFMSL{<q>}.F16, <Sn>, <Sm>[<index>]

128-bit SIMD vector (Q == 1)

VFMSL{<q>}.F16, <Dn>, <Dm>[<index>]

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' & & Vd<8> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';

T1 (Armv8.2)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>1</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
64-bit SIMD vector (Q == 0)

VFMSL{<q>}.F16 <Qd>, <Sn>, <Sm>[<index>]

128-bit SIMD vector (Q == 1)

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]

if InitBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';

Assembler Symbols

See *Standard assembler syntax fields*.

<q> Is the 128-bit name of the SIM&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qd> Is the 128-bit name of the first SIM&FP source register, encoded in the "N:Vn" field.

<Dn> Is the 64-bit name of the second SIM&FP source register, encoded in the "Vm<2:0>" field.

<Dm> Is the 64-bit name of the SIM&FP destination register, encoded in the "D:Vd" field.

<Sn> Is the 32-bit name of the first SIM&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIM&FP source register, encoded in the "Vm<2:0>:M" field.

<index> For the 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field.
For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation

CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;

element2 = Elem[operand2, index, esize DIV 2];
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, StandardFPSCRVal);
    D[d+r] = result;
**VFNMA**

Vector Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition. Depending on settings in the `CPACR`, `NSACR`, `HCPTR`, and `FPVEC` registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

**Half-precision scalar (size == 01)**

(Armv8.2)

VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

**Single-precision scalar (size == 10)**

VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

**Double-precision scalar (size == 11)**

If `FPSCR.Len != '000' || FPSCR.Stride != '00'` then **UNDEFINED**;
if `size == '00'` || (size == '01' && !HaveFP16Ext()) then **UNDEFINED**;
if `size == '01'` && `cond != '1110'` then **UNPREDICTABLE**;
`op1_neg = (op == '1');`
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If `size == '01'` && `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Half-precision scalar (size == 01)

(VFNMA\{\langle c\rangle\}\{\langle q\rangle\}.F16 \langle Sd\rangle, \langle Sn\rangle, \langle Sm\rangle)

Single-precision scalar (size == 10)

(VFNMA\{\langle c\rangle\}\{\langle q\rangle\}.F32 \langle Sd\rangle, \langle Sn\rangle, \langle Sm\rangle)

Double-precision scalar (size == 11)

(VFNMA\{\langle c\rangle\}\{\langle q\rangle\}.F64 \langle Dd\rangle, \langle Dn\rangle, \langle Dm\rangle)

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
op1_neg = (op == '1');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONstrained UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

\(<c>\) See Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<Sd>\) Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
\(<Sn>\) Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
\(<Sm>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  case esize of
    when 16
      op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
      S[d] = Zerops(16) \(\times\) FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, FPSCR[]);
    when 32
      op32 = if op1_neg then FPNeg(S[n]) else S[n];
      S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR[]);
    when 64
      op64 = if op1_neg then FPNeg(D[n]) else D[n];
      D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR[]);
VFNMS

Vector Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| != 1111 |
| 1 | 1 | 1 | 0 | 1 | D | 0 | 1 | Vn | Vd | 1 | 0 | size | N | 0 | M | 0 | Vm |
| cond | op

Half-precision scalar (size == 01) (Armv8.2)

VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
op1_neg = (op == '1');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | D | 0 | 1 | Vn | Vd | 1 | 0 | size | N | 0 | M | 0 | Vm |
| op
Half-precision scalar (size == 01)
(Armv8.2)
VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
op1_neg = (op == '1');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONstrained UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>    See Standard assembler syntax fields.
<q>    See Standard assembler syntax fields.
<Sd>   Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>   Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>   Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>   Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>   Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>   Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
    S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, FPSCR[]);
  when 32
    op32 = if op1_neg then FPNeg(S[n]) else S[n];
    S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR[]);
  when 64
    op64 = if op1_neg then FPNeg(D[n]) else D[n];
    D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR[]);
VHADD

Vector Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. For rounded results, see VRHADD).

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 1  | U  | 0  | D  | size | Vn | Vd | 0  | 0  | 0  | 0  | N  | Q  | M  | 0  | Vm | op  |

64-bit SIMD vector (Q == 0)

VHADD\{<c>\}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

128-bit SIMD vector (Q == 1)

VHADD\{<c>\}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0'); unsigned = (U == '1');
esize = 8 << Uint(size); elements = 64 DIV esize;
d = Uint(D:Vd); n = Uint(N:Vn); m = Uint(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | U  | 1  | 1  | 1  | 0  | D  | size | Vn | Vd | 0  | 0  | 0  | 0  | N  | Q  | M  | 0  | Vm | op  |

64-bit SIMD vector (Q == 0)

VHADD\{<c>\}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

128-bit SIMD vector (Q == 1)

VHADD\{<c>\}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0'); unsigned = (U == '1');
esize = 8 << Uint(size); elements = 64 DIV esize;
d = Uint(D:Vd); n = Uint(N:Vn); m = Uint(M:Vm); regs = if Q == '0' then 1 else 2;

**Assembler Symbols**

\(<c>\) For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
Is the data type for the elements of the operands, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Int(Elem[D[n+r],e,esize], unsigned);
      op2 = Int(Elem[D[m+r],e,esize], unsigned);
      result = if add then op1+op2 else op1-op2;
      Elem[D[d+r],e,esize] = result<esize:1>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Vector Halving Subtract subtracts the elements of the second operand from the corresponding elements of the first operand, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. There is no rounding version.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).
Is the data type for the elements of the operands, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U size</th>
</tr>
</thead>
</table>
| 0 00   | S8  
| 0 01   | S16 
| 0 10   | S32 
| 1 00   | U8  
| 1 01   | U16 
| 1 10   | U32 

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector move Insertion. This instruction copies the lower 16 bits of the 32-bit source SIMD&FP register into the upper 16 bits of the 32-bit destination SIMD&FP register, while preserving the values in the remaining bits. Depending on settings in the \textit{CPACR}, \textit{NSACR}, \textit{HCPTR}, and \textit{FPEXC} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support}.

It has encodings from the following instruction sets: A32 (\textit{A1}) and T32 (\textit{T1}).

A1 (Armv8.2)

\begin{verbatim}
\textbf{A1}
\end{verbatim}

\begin{verbatim}
VINS{<q>}.F16 <Sd>, <Sm>
if !\textit{HaveFP16Ext}() then UNDEFINED;
if \textit{FPSCR.Len} != '000' || \textit{FPSCR.Stride} != '00' then UNDEFINED;
d = \textit{UInt}(Vd:D); m = \textit{UInt}(Vm:M);
\end{verbatim}

T1 (Armv8.2)

\begin{verbatim}
\textbf{T1}
\end{verbatim}

\begin{verbatim}
VINS{<q>}.F16 <Sd>, <Sm>
if \textit{InITBlock}() then UNPREDICTABLE;
if !\textit{HaveFP16Ext}() then UNDEFINED;
if \textit{FPSCR.Len} != '000' || \textit{FPSCR.Stride} != '00' then UNDEFINED;
d = \textit{UInt}(Vd:D); m = \textit{UInt}(Vm:M);
\end{verbatim}

**CONSTRAINED UNPREDICTABLE behavior**

If \textit{InITBlock}(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

\textbf{Assembler Symbols}

- \textit{<q>} See \textit{Standard assembler syntax fields}.
- \textit{<Sd>} Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- \textit{<Sm>} Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

\textbf{Operation}

\begin{verbatim}
if \textit{ConditionPassed}() then
    EncodingSpecificOperations(); \textit{CheckVFPEnabled}(TRUE);
    \textit{S}[d] = \textit{S}[m]<15:0> : \textit{S}[d]<15:0>;
\end{verbatim}
VJCVT

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 SIMD&FP destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo $2^{32}$, as held in a 32-bit signed integer.

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

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1 (Armv8.3)

```
! = 1111 | 1 1 1 0 1 | D | 1 1 1 0 0 1 | Vd | 1 0 1 1 1 1 | M | 0 | Vm
```

```
cond
```

#### A1

VJCVT\(<q>\).S32.F64 <Sd>, <Dm>

if !HaveFJCVTZSExt() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
\nd = Uint(Vd:D); \nm = Uint(M:Vm);

#### T1 (Armv8.3)

```
1 1 1 0 1 1 1 0 1 | D | 1 1 1 0 0 1 | Vd | 1 0 1 1 1 1 | M | 0 | Vm
```

```
VJCVT<q>\).S32.F64 <Sd>, <Dm>
```

if !HaveFJCVTZSExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
\nd = Uint(Vd:D); \nm = Uint(M:Vm);

### Assembler Symbols

- `<q>` See Standard assembler syntax fields.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation

EncodingSpecificOperations();

CheckVFPEndabled(TRUE);

\n\nbits(64) fltval = D[m];
bits(32) intval;
bit Z;

\n\n(intval, Z) = FPtoFixedJS(fltval, FPSCR[], FALSE);
FPSCR<31:28> = '0':Z:'00';
S[d] = intval;
VLD1 (single element to one lane)

Load single 1-element structure to one lane of one register loads one element from memory into one element of a register. Elements of the register that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1, A2 and A3) and T32 (T1, T2 and T3).

**A1**

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

**Offset (Rm == 1111)**

VLD1{<c>}{<q>}.{<size> <list>, [<Rn>{:<align}>]

**Post-indexed (Rm == 1101)**

VLD1{<c>}{<q>}.{<size> <list>, [<Rn>{:<align}>]}

**Post-indexed (Rm != 11x1)**

VLD1{<c>}{<q>}.{<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then SEE “VLD1 (single element to all lanes)”;  
if index_align<0> != '0' then UNDEFINED;  
ebytes = 1; index = UInt(index_align<3:1>); alignment = 1;  
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);  
 wback = (m != 15); register_index = (m != 15 && m != 13);  
if n == 15 then UNPREDICTABLE;

**A2**

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

Post-indexed (Rm != 11x1)
Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<1> != '0' then UNDEFINED;
  ebytes = 2;  index = UInt(index_align<3:2>);
  alignment = if index_align<0> == '0' then 1 else 2;
  d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
  wback = (m != 15);  register_index = (m != 15 && m != 13);
  if n == 15 then UNPREDICTABLE;

T1

A3

VLD1 (single element to one lane)
Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then SEE "VLD1 (single element to all lanes)"
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

T2

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 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 D 1 0 Rn  Vd 0 1 0 0 index_align  Rm  

size

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then SEE "VLD1 (single element to all lanes)"
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

T3

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 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 D 1 0 Rn  Vd 1 0 0 0 index_align  Rm  

size
Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

if size == '11' then SEE “VLD1 (single element to all lanes)”; if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4; index = Uint(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = Uint(D:Vd); n = Uint(Rn); m = Uint(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> For encoding A1, A2 and A3: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1, T2 and T3: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> Is the data size, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
</tbody>
</table>

<list> Is a list containing the single 64-bit name of the SIMD&FP register holding the element.
The list must be { <Dd>[<index>] }. The register <Dd> is encoded in the “D:Vd” field.
The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.

<size> == 16
<index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.

<size> == 32
<index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn> Is the general-purpose base register, encoded in the “Rn” field.

<align> When <size> == 8, <align> must be omitted, otherwise it is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on <size>:

<size> == 8
Encoded in the “index_align<0>” field as 0.

<size> == 16
Encoded in the “index_align<1:0>” field as 0b00.

<size> == 32
Encoded in the “index_align<2:0>” field as 0b000.

Whenever <align> is present, the permitted values and encoding depend on <size>:
<size> == 16
<align> is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

<size> == 32
<align> is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as 0b011.

is is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDEnabled();
   address = R[n]; iswrite = FALSE;
   - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
   Elem[D[d],index] = MemU[address,ebytes];
   if wback then
      if register_index then
         R[n] = R[n] + R[m];
      else
         R[n] = R[n] + ebytes;
   
Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VLD1 (single element to all lanes)

Load single 1-element structure and replicate to all lanes of one register loads one element from memory into every element of one or two vectors. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>],<Rm>

if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wbback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 0 0 1 1</td>
</tr>
</tbody>
</table>
Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<<align>}}]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<<align}>]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<<align}>], <Rm>

if size == '11' || (size == '00' && a == '1') then UNDEFINED;

ebytes = 1 << UInt(size);  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD1 (single element to all lanes).

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> Is the data size, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<list> Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:

{ <Dd>[ ] }  
Encoded in the "T" field as 0.

{ <Dd>[ ], <Dd+1>[ ] }  
Encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> When <size> == 8, <align> must be omitted, otherwise it is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "a" field as 0.
Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16
<align> is 16, meaning 16-bit alignment, encoded in the "a" field as 1.

<size> == 32
<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.
<Rm> is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    bits(64) replicated_element = Replicate(MemU[address,ebytes]);
    for r = 0 to regs-1
        D[d+r] = replicated_element;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VLD1 (multiple single elements)

Load multiple single 1-element structures to one, two, three, or four registers loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, A3 and A4) and T32 (T1, T2, T3 and T4).

A1

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

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

reg = 1; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 0 0</td>
</tr>
</tbody>
</table>
Offset (Rm == 1111)
VLD1{:<c>}{:<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)
VLD1{:<c>}{:<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)
VLD1{:<c>}{:<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

regs = 2; if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << Uint(align);
ebytes = 1 << Uint(size); elements = 8 DIV ebytes;
d = Uint(D:Vd); n = Uint(Rn); m = Uint(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

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

Offset (Rm == 1111)
VLD1{:<c>}{:<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)
VLD1{:<c>}{:<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)
VLD1{:<c>}{:<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

regs = 3; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << Uint(align);
ebytes = 1 << Uint(size); elements = 8 DIV ebytes;
d = Uint(D:Vd); n = Uint(Rn); m = Uint(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A4
Offset (Rm == 1111)

\[ \text{VLD1}\{\langle c\rangle}\{\langle q\rangle}\}.\langle \text{size} \rangle \langle \text{list} \rangle, \{\langle Rn\rangle\{\langle \text{align} \rangle\} \]

Post-indexed (Rm == 1101)

\[ \text{VLD1}\{\langle c\rangle}\{\langle q\rangle\}.\langle \text{size} \rangle \langle \text{list} \rangle, \{\langle Rn\rangle\{\langle \text{align} \rangle\} \]

Post-indexed (Rm != 11x1)

\[ \text{VLD1}\{\langle c\rangle}\{\langle q\rangle\}.\langle \text{size} \rangle \langle \text{list} \rangle, \{\langle Rn\rangle\{\langle \text{align} \rangle\}, \langle Rm\rangle \]

\begin{align*}
\text{regs} & = 4; \\
\text{alignment} & = \text{if align} == '00' \text{ then } 1 \text{ else } 4 \text{ } \ll \text{ UInt}(\text{align}); \\
\text{cbytes} & = 1 \text{ } \ll \text{ UInt}(\text{size}); \text{ elements} = 8 \text{ DIV } \text{cbytes}; \\
\text{d} & = \text{ UInt}(D:Vd); \text{ } n = \text{ UInt}(Rn); \text{ } m = \text{ UInt}(Rm); \\
\text{wback} & = (m != 15); \text{ register_index} = (m != 15 \&\& m != 13); \\
\text{if } n == 15 \text{ || d+regs > 32 then UNPREDICTABLE; }
\end{align*}

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

\begin{align*}
\text{15} & \text{ 14} \text{ 13} \text{ 12} \text{ 11} \text{ 10} \text{ 9} \text{ 8} \text{ 7} \text{ 6} \text{ 5} \text{ 4} \text{ 3} \text{ 2} \text{ 1} \text{ 0} \\
\text{1} & \text{ 1} \text{ 1} \text{ 1} \text{ 1} \text{ 0} \text{ 0} \text{ 1} \text{ 0} \text{ D} \text{ 1} \text{ 0} \text{ Rn} \text{ Vd} \text{ 0} \text{ 1} \text{ 1} \text{ 1} \text{ size} \text{ align} \text{ Rm}
\end{align*}

Offset (Rm == 1111)

\[ \text{VLD1}\{\langle c\rangle}\{\langle q\rangle\}.\langle \text{size} \rangle \langle \text{list} \rangle, \{\langle Rn\rangle\{\langle \text{align} \rangle\} \]

Post-indexed (Rm == 1101)

\[ \text{VLD1}\{\langle c\rangle}\{\langle q\rangle\}.\langle \text{size} \rangle \langle \text{list} \rangle, \{\langle Rn\rangle\{\langle \text{align} \rangle\} \]

Post-indexed (Rm != 11x1)

\[ \text{VLD1}\{\langle c\rangle}\{\langle q\rangle\}.\langle \text{size} \rangle \langle \text{list} \rangle, \{\langle Rn\rangle\{\langle \text{align} \rangle\}, \langle Rm\rangle \]

\begin{align*}
\text{regs} & = 1; \text{ if align<1> == '1' then UNDEFINED; } \\
\text{alignment} & = \text{if align == '00' then 1 else } 4 \text{ } \ll \text{ UInt}(\text{align}); \\
\text{cbytes} & = 1 \text{ } \ll \text{ UInt}(\text{size}); \text{ elements} = 8 \text{ DIV } \text{cbytes}; \\
\text{d} & = \text{ UInt}(D:Vd); \text{ } n = \text{ UInt}(Rn); \text{ } m = \text{ UInt}(Rm); \\
\text{wback} & = (m != 15); \text{ register_index} = (m != 15 \&\& m != 13); \\
\text{if } n == 15 \text{ || d+regs > 32 then UNPREDICTABLE; }
\end{align*}

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

---

**T2**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Offset (Rm == 1111)**

VLD1{{<c>}{<q>../../../<list>}, [{<Rn}>{:<align}>]}

**Post-indexed (Rm == 1101)**

VLD1{{<c>}{<q>../../../<list>}, [{<Rn}>{:<align}>]}

**Post-indexed (Rm != 11x1)**

VLD1{{<c>}{<q>../../../<list>}, [{<Rn}>{:<align}>]}, <Rm>

regs = 2; if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

---

**CONSTRANGED UNPREDICTABLE behavior**

If d+<regs> > 32, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

---

**T3**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>0</td>
</tr>
</tbody>
</table>

**Offset (Rm == 1111)**

VLD1{{<c>}{<q>../../../<list>}, [{<Rn}>{:<align}>]}

**Post-indexed (Rm == 1101)**

VLD1{{<c>}{<q>../../../<list>}, [{<Rn}>{:<align}>]}

**Post-indexed (Rm != 11x1)**

VLD1{{<c>}{<q>../../../<list>}, [{<Rn}>{:<align}>]}, <Rm>

regs = 3; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

---

VLD1 (multiple single elements)
CONSTRANDED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T4

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 1 0 0 1 0 | D 1 0 | Rn | Vd | 0 0 1 0 | size | align | Rm |

Offset (Rm == 1111)

VLD1{<c>{<q>}.<size> <list>, [<Rn>{:<align}>]}

Post-indexed (Rm == 1101)

VLD1{<c>{<q>}.<size> <list>, [<Rn>{:<align}>]!}

Post-indexed (Rm != 11x1)

VLD1{<c>{<q>}.<size> <list>, [<Rn>{:<align}>]}, <Rm>

regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
writeback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD1 (multiple single elements).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

- For encoding A1, A2, A3 and A4: see Standard assembler syntax fields. This encoding must be unconditional.
- For encoding T1, T2, T3 and T4: see Standard assembler syntax fields.

- See Standard assembler syntax fields.

- Is the data size, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>64</td>
</tr>
</tbody>
</table>

- Is a list containing the 64-bit names of the SIMD&FP registers.
  The list must be one of:
Single register. Selects the A1 and T1 encodings of the instruction.

Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.

Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.

Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as 0b01.

128 128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.

256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            bits(ebytes*8) data;
            if ebytes != 8 then
                data = MemU[address,ebytes];
                else
                    data = MemU[address+4,4] else MemU[address,4];
            data<31:0> = if BigEndian(AccType_NORMAL) then MemU[address+4,4] else MemU[address,4];
            data<63:32> = if BigEndian(AccType_NORMAL) then MemU[address,4] else MemU[address+4,4];
            Elem[D[d+r],e] = data;
            address = address + ebytes;
        if wback then
            if register_index then
                R[n] = R[n] + R[m];
            else
                R[n] = R[n] + 8*regs;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VLD2 (single 2-element structure to one lane)

Load single 2-element structure to one lane of two registers loads one 2-element structure from memory into corresponding elements of two registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2 and A3) and T32 (T1, T2 and T3).

A1

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

Offset (Rm == 1111)

VLD2{<c>}{<q>}.${size} <list>, [<Rn>{:<align}]}

Post-indexed (Rm == 1101)

VLD2{<c>}{<q>}.${size} <list>, [<Rn>{:<align}]!

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q>}.${size} <list>, [<Rn>{:<align}]}, <Rm>

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";

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

CONstrained UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.
Offset (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wbback = (m != 15);  register_index = (m != 15 & m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

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

Offset (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wbback = (m != 15);  register_index = (m != 15 & m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 0 0 1 1 | D 1 0 | Rn | Vd | 0 0 0 1 | index_align | Rm |
| size |

Offset (Rm == 1111)

VLD2{<c>{<q}>.<size> <list>, [<Rn>{:<align>}]}

Post-indexed (Rm == 1101)

VLD2{<c>{<q}>.<size> <list>, [<Rn>{:<align}>]!}

Post-indexed (Rm != 11x1)

VLD2{<c>{<q}>.<size> <list>, [<Rn>{:<align}>], <Rm>}

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)"

ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONstrained unpredictable behavior

If d2 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 0 0 1 1 | D 1 0 | Rn | Vd | 0 1 0 1 | index_align | Rm |
| size |

Offset (Rm == 1111)

VLD2{<c>{<q}>.<size> <list>, [<Rn>{:<align>}]}

Post-indexed (Rm == 1101)

VLD2{<c>{<q}>.<size> <list>, [<Rn>{:<align}>]!}

Post-indexed (Rm != 11x1)

VLD2{<c>{<q}>.<size> <list>, [<Rn>{:<align}>], <Rm>}

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)"

ebytes = 2; index = UInt(index_align<3:2>); inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
CONCONSTRAINED UNPREDICTABLE behavior

If \( d_2 > 31 \), then one of the following behaviors must occur:

- The instruction is \textbf{UNDEFINED}.
- The instruction executes as \textbf{NOP}.
- One or more of the SIMD and floating-point registers are \textbf{UNKNOWN}. If the instruction specifies writeback, the base register becomes \textbf{UNKNOWN}. This behavior does not affect any general-purpose registers.

T3

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & D & 1 & 0 & Rn & Vd & 1 & 0 & 0 & 1 & index\_align & Rm
\end{array}
\]

size

Offset (Rm == 1111)

\[
\text{VLD2}\{<c>\}{<q>\}.<size> <list>, [<Rn>{:<align}>]}
\]

Post-indexed (Rm == 1101)

\[
\text{VLD2}\{<c>\}{<q>\}.<size> <list>, [<Rn>{:<align}>]!}
\]

Post-indexed (Rm != 11x1)

\[
\text{VLD2}\{<c>\}{<q>\}.<size> <list>, [<Rn>{:<align}>], <Rm>
\]

\[
\begin{align*}
\text{if size == '11' then SEE “VLD2 (single 2-element structure to all lanes)”;} \\
\text{if index\_align<1> != '0' then UNDEFINED;} \\
\text{ebytes = 4; index = UInt(index\_align<3>);} \\
\text{inc = if index\_align<2> == '0' then 1 else 2;} \\
\text{alignment = if index\_align<0> == '0' then 1 else 8;} \\
\text{d = UInt(D:\text{Vd}); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);} \\
\text{wback = (m != 15); register\_index = (m != 15 & m != 13);} \\
\text{if n == 15 || d2 > 31 then UNPREDICTABLE;}
\end{align*}
\]

CONCONSTRAINED UNPREDICTABLE behavior

If \( d_2 > 31 \), then one of the following behaviors must occur:

- The instruction is \textbf{UNDEFINED}.
- The instruction executes as \textbf{NOP}.
- One or more of the SIMD and floating-point registers are \textbf{UNKNOWN}. If the instruction specifies writeback, the base register becomes \textbf{UNKNOWN}. This behavior does not affect any general-purpose registers.

For more information about the \textbf{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see \textit{Architectural Constraints on UNPREDICTABLE behaviors}, and particularly \textit{VLD2 (single 2-element structure to one lane)}.

Assembler Symbols

\texttt{<c>}

For encoding A1, A2 and A3: see \textit{Standard assembler syntax fields}. This encoding must be unconditional.

For encoding T1, T2 and T3: see \textit{Standard assembler syntax fields}.

\texttt{<q>}

See \textit{Standard assembler syntax fields}.

\texttt{<size>}

Is the data size, encoded in “size”:

\[
\begin{array}{c|c}
\text{size} & <\text{size}> \\
00 & 8 \\
01 & 16 \\
10 & 32 \\
\end{array}
\]

\texttt{<list>}

Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.

The list must be one of:
{ <Dd>[<index>], <Dd+1>[<index>] }  
Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>] }  
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:
<size> == 16
"spacing" is encoded in the "index_align<1>" field.

<size> == 32
"spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:
<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn>  Is the general-purpose base register, encoded in the "Rn" field.

<align>  Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on <size>:
<size> == 8
Encoded in the "index_align<0>" field as 0.

<size> == 16
Encoded in the "index_align<0>" field as 0.

<size> == 32
Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:
<size> == 8
<align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16
<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32
<align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm>  Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
VLD2 (single 2-element structure to all lanes)

Load single 2-element structure and replicate to all lanes of two registers loads one 2-element structure from memory into all lanes of two registers. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

Offset (Rm == 1111)

VLD2{<c>}{<q}>.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD2{<c>}{<q}>.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q}>.<size> <list>, [<Rn>{:<align}>],<Rm>

if size == '11' then UNDEFINED;
ebytes = 1 << Uint(size);
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = Uint(D:Vd);  d2 = d + inc;  n = Uint(Rn);  m = Uint(Rm);
Wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 1 1 0 0 1 1 | D | 1 | 0 | Rn | Vd | 1 1 0 1 | size | T | a | Rm |
Offset \((Rm == 1111)\)

\[
\text{VLD2}\{<c>{}<q>}.{<size> <list>}, [<Rn>:{<align>}]\}
\]

Post-indexed \((Rm == 1101)\)

\[
\text{VLD2}\{<c>{}<q>}.{<size> <list>}, [<Rn>:{<align>}]\}!
\]

Post-indexed \((Rm != 11x1)\)

\[
\text{VLD2}\{<c>{}<q>}.{<size> <list>}, [<Rn>:{<align>}], <Rm>\}
\]

\[
\begin{align*}
\text{if size} & == '11' \text{ then UNDEFINED;} \\
ebytes & = 1 << \text{UInt}(size); \\
alignment & = \text{if a} == '0' \text{ then 1 else 2*ebytes;} \\
inc & = \text{if T} == '0' \text{ then 1 else 2;} \\
d & = \text{UInt}(D:Vd); \quad d2 = d + inc; \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \\
wback & = (m != 15); \quad \text{register_index} = (m != 15 \&\& m != 13); \\
\text{if n} & == 15 \mid d2 > 31 \text{ then UNPREDICTABLE};
\end{align*}
\]

**CONSTRUDED UNPREDICTABLE behavior**

If \(d2 > 31\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRUDED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD2 (single 2-element structure to all lanes)*.

**Assembler Symbols**

- \(<c>\) For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
- For encoding T1: see *Standard assembler syntax fields*.
- \(<q>\) See *Standard assembler syntax fields*.
- \(<size>\) Is the data size, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- \(<list>\) Is a list containing the 64-bit names of two SIMD&FP registers.
- The list must be one of:
  - \(\{ <Dd>[], <Dd+1>[] \}\)
    Single-spaced registers, encoded in the “T” field as 0.
  - \(\{ <Dd>[], <Dd+2>[] \}\)
    Double-spaced registers, encoded in the “T” field as 1.
- The register \(<Dd>\) is encoded in the "D:Vd" field.
- \(<Rn>\) Is the general-purpose base register, encoded in the “Rn” field.
- \(<align>\) Is the optional alignment.
  - Whenever \(<align>\) is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "a" field as 0.
  - Whenever \(<align>\) is present, the permitted values and encoding depend on \(<size>\):
    - \(<size> == 8\)
      - \(<align>\) is 16, meaning 16-bit alignment, encoded in the "a" field as 1.
<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

<align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm> is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
    else
        R[n] = R[n] + 2*ebytes;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VLD2 (multiple 2-element structures)

Load multiple 2-element structures to two or four registers loads multiple 2-element structures from memory into two or four registers, with de-interleaving. For more information, see Element and structure load/store instructions. Every element of each register is loaded. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

| 31 30 29 28 27 26 25 24 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 | 1 | 0 | 0 | 0 | D | 1 | 0 | Rn | Vd | 1 | 0 | 0 | x | size | align | Rm |

itype

Offset (Rm == 1111)

VLD2<{c}>{q}.<size> <list>, [{Rn}{:<align>}]

Post-indexed (Rm == 1101)

VLD2<{c}>{q}.<size> <list>, [{Rn}{:<align}>]!

Post-indexed (Rm != 11x1)

VLD2<{c}>{q}.<size> <list>, [{Rn}{:<align}>], {Rm}

regs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm); wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d2+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

| 31 30 29 28 27 26 25 24 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 | 1 | 0 | 0 | 0 | D | 1 | 0 | Rn | Vd | 0 | 0 | 1 | 1 | size | align | Rm |
Offset (Rm == 1111)

\[
\text{VLD2}<c>{!<q>}.<size> <list>, [<Rn}{:<align}>]
\]

Post-indexed (Rm == 1101)

\[
\text{VLD2}<c>{!<q>}.<size> <list>, [<Rn}{:<align}>]!
\]

Post-indexed (Rm != 11x1)

\[
\text{VLD2}<c>{!<q>}.<size> <list>, [<Rn}{:<align}>], <Rm>
\]

\[
\text{regs} = 2; \quad \text{inc} = 2;
\]

\[
\text{if size == '11' then UNDEFINED;}
\]

\[
\text{alignment = if align == '00' then 1 else 4 << UInt(align);} \\
\text{ebytes = 1 << UInt(size); \quad elements = 8 DIV ebytes;} \\
\text{d = UInt(D:Vd); \quad d2 = d + inc; \quad n = UInt(Rn); \quad m = UInt(Rm);} \\
\text{wback = (m != 15); \quad register_index = (m != 15 && m != 13);} \\
\text{if n == 15 || d2+regs > 32 then UNPREDICTABLE;}
\]

CONSTRUED UNPREDICTABLE behavior

If d2+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

\[
\begin{array}{cccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & D & 1 & 0 & Rn & Vd & 1 & 0 & 0 & x & \text{size} & \text{align} & \text{Rm} \\
\end{array}
\]

itype

Offset (Rm == 1111)

\[
\text{VLD2}<c>{!<q>}.<size> <list>, [<Rn}{:<align}>]
\]

Post-indexed (Rm == 1101)

\[
\text{VLD2}<c>{!<q>}.<size> <list>, [<Rn}{:<align}>]!
\]

Post-indexed (Rm != 11x1)

\[
\text{VLD2}<c>{!<q>}.<size> <list>, [<Rn}{:<align}>], <Rm>
\]

\[
\text{regs} = 1; \quad \text{if align == '11' then UNDEFINED;} \\
\text{if size == '11' then UNDEFINED;} \\
\text{inc = if itype == '1001' then 2 else 1;} \\
\text{alignment = if align == '00' then 1 else 4 << UInt(align);} \\
\text{ebytes = 1 << UInt(size); \quad elements = 8 DIV ebytes;} \\
\text{d = UInt(D:Vd); \quad d2 = d + inc; \quad n = UInt(Rn); \quad m = UInt(Rm);} \\
\text{wback = (m != 15); \quad register_index = (m != 15 && m != 13);} \\
\text{if n == 15 || d2+regs > 32 then UNPREDICTABLE;}
\]

CONSTRUED UNPREDICTABLE behavior

If d2+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2

Offset (Rm == 1111)

VLD2{<c>}{<q>}{<size> <list>}, [<Rn>{:<align>}]}

Post-indexed (Rm == 1101)

VLD2{<c>}{<q>}{<size> <list>}, [<Rn>{:<align>}]}

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q>}{<size> <list>}, [<Rn>{:<align>}], <Rm>

regs = 2; inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d2+regs > 32, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD2 (multiple 2-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

<c>
For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<size>
Is the data size, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<list>
Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:

{ <Dd>, <Dd+1> }
Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the “itype” field as 0b1000.
{ <Dd>, <Dd+2> }
Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in
the "itype" field as 0b1001.

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }
Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and is
encoded in the "align" field as 0b00.
Whenever <align> is present, the permitted values are:

64  64-bit alignment, encoded in the "align" field as 0b01.
128 128-bit alignment, encoded in the "align" field as 0b10.
256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four
     registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>,
see Advanced SIMD addressing mode.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm"
field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r], e] = MemU[address,ebytes];
            Elem[D2[d2+r],e] = MemU[address+ebytes,ebytes];
            address = address + 2*ebytes;
        if wback then
            if register_index then
                R[n] = R[n] + R[m];
            else
                R[n] = R[n] + 16*regs;
    

VLD3 (single 3-element structure to one lane)

Load single 3-element structure to one lane of three registers loads one 3-element structure from memory into corresponding elements of three registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2 and A3) and T32 (T1, T2 and T3).

A1

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

Offset (Rm == 1111)

VLD3{<c>}{<q>}{<size> <list>, [<Rn]}

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}{<size> <list>, [<Rn]}!

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}{<size> <list>, [<Rn]}, <Rm>

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0>= '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

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

VLD3 (single 3-element structure to one lane)
Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn]}

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn]!}

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn]}

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)"
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

```
|   |   |   |   |   |   |   |   | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | D | 1 | 0 |    |    |    |    |    |    |    |    |    |    |    |    |
|   |   |   |   |   |   |   |   | size|   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn]}

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn]!}

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn]}

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)"
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.
Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn]}

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 111x)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)"
if index_align<0> != '0' then UNDEFINED;
  ebytes = 2;  index = UFix(index_align<3:2>);
  inc = if index_align<1> == '0' then 1 else 2;
  d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
  wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONstrained UNPredictable behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn]}

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 111x)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)"
if index_align<0> != '0' then UNDEFINED;
  ebytes = 1;  index = UFix(index_align<3:1>);
  inc = 1;
  d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
  wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
CONSTRAINED UNPREDICTABLE behavior

If \( d3 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

### T3

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

index_align | Rm
---|---
size

**Offset (Rm == 1111)**

\[ \text{VLD3}\{<c>\}{<q>}\{<size> <list>, [<Rn]>} \]

**Post-indexed (Rm == 1101)**

\[ \text{VLD3}\{<c>\}{<q>}\{<size> <list>, [<Rn>]!} \]

**Post-indexed (Rm != 11x1)**

\[ \text{VLD3}\{<c>\}{<q>}\{<size> <list>, [<Rn>], <Rm>} \]

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)"
if index_align<1:0> != '00' then UNDEFINED;
bytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15) & (m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( d3 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD3 (single 3-element structure to one lane).

**Assembler Symbols**

- `<c>` For encoding A1, A2 and A3: see Standard assembler syntax fields. This encoding must be unconditional.
  For encoding T1, T2 and T3: see Standard assembler syntax fields.

- `<q>` See Standard assembler syntax fields.

- `<size>` Is the data size, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
</tbody>
</table>

- `<list>` Is a list containing the 64-bit names of the three SIMD&FP registers holding the element. The list must be one of:
The encoding of "spacing" depends on <size>:

<size> == 8
"spacing" is encoded in the "index_align<0>" field.

<size> == 16
"spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.

<size> == 32
"spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Alignment
Standard alignment rules apply, see Alignment support.

Operation

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDEnabled();
   address = R[n];
   Elem[D[d], index] = MemU[address,ebytes];
   Elem[D[d2],index] = MemU[address+ebytes,ebytes];
   Elem[D[d3],index] = MemU[address+2*ebytes,ebytes];
   if wback then
      if register_index then
         R[n] = R[n] + R[m];
      else
         R[n] = R[n] + 3*ebytes;
   else
   end

VLD3 (single 3-element structure to one lane)
VLD3 (single 3-element structure to all lanes)

Load single 3-element structure and replicate to all lanes of three registers loads one 3-element structure from memory into all lanes of three registers. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>!]

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRANED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | D  | 1  | 0  | Rn | Vd | 1  | 1  | 1  | 0  | size | T | 0 | Rm |
Offset (Rm == 1111)

VLD3{<c>}{<q>}{<size>} <list>, [<Rn>]

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}{<size>} <list>, [<Rn>]

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}{<size>} <list>, [<Rn>], <Rm>

if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD3 (single 3-element structure to all lanes).

Assembler Symbols

<c>  
For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q>  
See Standard assembler syntax fields.

<size>  
Is the data size, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

(list)  
Is a list containing the 64-bit names of three SIMD&FP registers.
The list must be one of:

{ <Dd>[1], <Dd+1>[1], <Dd+2>[1] }  
Single-spaced registers, encoded in the "T" field as 0.

{ <Dd>[1], <Dd+2>[1], <Dd+4>[1] }  
Double-spaced registers, encoded in the "T" field as 1.
The register <Dd> is encoded in the "D:Vd" field.

<Rn>  
Is the general-purpose base register, encoded in the "Rn" field.

<Rm>  
Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Alignment
Standard alignment rules apply, see Alignment support.
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n];
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes]);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
    else
        R[n] = R[n] + 3*ebytes;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VLD3 (multiple 3-element structures)

Load multiple 3-element structures to three registers loads multiple 3-element structures from memory into three registers, with de-interleaving. For more information, see Element and structure load/store instructions. Every element of each register is loaded. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

Offset (Rm == 1111)

VLD3{<c>{<q>}}.<size> <list>, [<<Rn>{{<align>}}]

Post-indexed (Rm == 1101)

VLD3{<c>{<q>}}.<size> <list>, [<<Rn>{{<align>}}]!

Post-indexed (Rm != 11x1)

VLD3{<c>{<q>}}.<size> <list>, [<<Rn>{{<align>}}]}, <Rm>

case iype of
  when '0100' 
    inc = 1;
  when '0101' 
    inc = 2;
  otherwise
    SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15) && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRUED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | D | 1 | 0 | Rn | Vd | 0 | 1 | 0 | x | size | align | Rm |
| iype |
Offset (Rm == 1111)

\[
\text{VLD3}\{\langle c\rangle}\{\langle q\rangle\}.\langle\text{size}\rangle\{\langle\text{list}\rangle\}, \{\langle Rn\rangle\{\langle\text{align}\rangle\}\}
\]

Post-indexed (Rm == 1101)

\[
\text{VLD3}\{\langle c\rangle}\{\langle q\rangle\}.\langle\text{size}\rangle\{\langle\text{list}\rangle\}, \{\langle Rn\rangle\{\langle\text{align}\rangle\}\}
\]

Post-indexed (Rm != 11x1)

\[
\text{VLD3}\{\langle c\rangle}\{\langle q\rangle\}.\langle\text{size}\rangle\{\langle\text{list}\rangle\}, \{\langle Rn\rangle\{\langle\text{align}\rangle\}\}
\]

case itype of
  when '0100'
    inc = 1;
  when '0101'
    inc = 2;
  otherwise
    SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << \text{UInt}(size);
elements = 8 DIV ebytes;
d = \text{UInt}(D:Vd); d2 = d + inc; d3 = d2 + inc; n = \text{UInt}(Rn); m = \text{UInt}(Rm);
wback = (m != 15); register_index = (m != 15 & & m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD3 (multiple 3-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

\(<c\>\) For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

\(<q\>\) See Standard assembler syntax fields.

\(<\text{size}\>\) Is the data size, encoded in "\text{size}"

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\text{size}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<\text{list}\>\) Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:

\{ \langle Dd\rangle, \langle Dd+1\rangle, \langle Dd+2\rangle \}  
Single-spaced registers, encoded in the "itype" field as 0b0100.

\{ \langle Dd\rangle, \langle Dd+2\rangle, \langle Dd+4\rangle \}  
Double-spaced registers, encoded in the "itype" field as 0b0101.

The register \langle Dd\rangle is encoded in the "D:Vd" field.

\(<Rn\>\) Is the general-purpose base register, encoded in the "Rn" field.
<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as 0b00. Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as 0b01. : is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about <Rn>, !, and <Rm>, see Advanced SIMD addressing mode.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address,ebytes];
        Elem[D[d2], e] = MemU[address+ebytes,ebytes];
        Elem[D[d3], e] = MemU[address+2*ebytes,ebytes];
        address = address + 3*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
VLD4 (single 4-element structure to one lane)

Load single 4-element structure to one lane of four registers loads one 4-element structure from memory into corresponding elements of four registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2 and A3) and T32 (T1, T2 and T3).

A1

\[
\begin{array}{cccccccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & D & 1 & 0 & Rn & Vd & 0 & 0 & 1 & 1 & index\_align & Rm \\
\end{array}
\]

Size

Offset (Rm == 1111)

\[
\text{VLD4} \{<c>\} \{<q>\}.<\text{size}> <\text{list}>, [<Rn>\{:<\text{align}>\}]
\]

Post-indexed (Rm == 1101)

\[
\text{VLD4} \{<c>\} \{<q>\}.<\text{size}> <\text{list}>, [<Rn>\{:<\text{align}>\}]!
\]

Post-indexed (Rm != 11x1)

\[
\text{VLD4} \{<c>\} \{<q>\}.<\text{size}> <\text{list}>, [<Rn>\{:<\text{align}>\}], <\text{Rm}>
\]

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";

ebytes = 1;  index = UInt(index\_align<3:1>);  inc = 1;

alignment = if index\_align<0> == '0' then 1 else 4;

d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);

wback = (m != 15);  register\_index = (m != 15 && m != 13);

if n == 15 || d4 > 31 then UNPREDICTABLE;

**CONstrained UNPREDICTABLE behavior**

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

\[
\begin{array}{cccccccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & D & 1 & 0 & Rn & Vd & 0 & 1 & 1 & 1 & index\_align & Rm \\
\end{array}
\]

Size
Offset (Rm == 1111)

VLD4{{<c>}{<q>}.<size> <list>, [{<Rn}>{:<align>}}]

Post-indexed (Rm == 1101)

VLD4{{<c>}{<q>}.<size> <list>, [{<Rn}>{:<align}>]}!

Post-indexed (Rm != 11x1)

VLD4{{<c>}{<q>}.<size> <list>, [{<Rn}>{:<align}>}, {<Rm>}

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wbback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRANED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

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

size

Offset (Rm == 1111)

VLD4{{<c>}{<q>}.<size> <list>, [{<Rn}>{:<align>}}]

Post-indexed (Rm == 1101)

VLD4{{<c>}{<q>}.<size> <list>, [{<Rn}>{:<align}>]}!

Post-indexed (Rm != 11x1)

VLD4{{<c>}{<q>}.<size> <list>, [{<Rn}>{:<align}>}, {<Rm>}

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 8 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wbback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRANED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | D | 1 | 0 |   |   |   |   |   | Rn |   |   | Vd |   | 0 | 0 | 1 | 1 | index_align |   | Rm |   |

**Offset (Rm == 1111)**

VLD4{{c}{{q}}.<size> <list>, [<Rn>{{<align>}}]}

**Post-indexed (Rm == 1101)**

VLD4{{c}{{q}}.<size> <list>, [<Rn>{{<align>}}]}!!

**Post-indexed (Rm != 11x1)**

VLD4{{c}{{q}}.<size> <list>, [<Rn>{{<align>}}]}, <Rm>

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);  inc = 1;
alignment = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

**CONstrained UNPredICTable behavior**

If d4 > 31, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

**T2**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 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 | D | 1 | 0 |   |   |   |   | Rn |   |   | Vd |   | 0 | 1 | 1 | 1 | index_align |   | Rm |   |

**Offset (Rm == 1111)**

VLD4{{c}{{q}}.<size> <list>, [<Rn>{{<align>}}]}

**Post-indexed (Rm == 1101)**

VLD4{{c}{{q}}.<size> <list>, [<Rn>{{<align>}}]}!!

**Post-indexed (Rm != 11x1)**

VLD4{{c}{{q}}.<size> <list>, [<Rn>{{<align>}}]}, <Rm>

if size == '11' then SEE "VLD4 (single 4-element structure to one lane)";
ebytes = 2;  index = UInt(index_align<3:2>);  inc = 1;
alignment = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
CONstrained UNPREDICTABLE behavior

If \( d4 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**T3**

```
  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
1 1 1 1 1 1 0 1 1 D 1 0  Rn  Vd  1 0 1 1  index_align  Rm
```

**Offset (Rm == 1111)**

\[ \text{VLD4}\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]} \]

**Post-indexed (Rm == 1101)**

\[ \text{VLD4}\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]}! \]

**Post-indexed (Rm != 11x1)**

\[ \text{VLD4}\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]}, <Rm> \]

```python
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
bytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m == 15); register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONstrained UNPREDICTABLE behavior

If \( d4 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD4 (single 4-element structure to one lane).

**Assembler Symbols**

- `<c>` For encoding A1, A2 and A3: see Standard assembler syntax fields. This encoding must be unconditional.
  For encoding T1, T2 and T3: see Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<size>` Is the data size, encoded in "size":
  ```
  size  <size>
  00  8
  01 16
  10 32
  ```
- `<list>` Is a list containing the 64-bit names of the four SIMD&FP registers holding the element. The list must be one of:

VLD4 (single 4-element structure to one lane)
{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] }  
  Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] }  
  Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 16  
  "spacing" is encoded in the "index_align<1>" field.

<size> == 32  
  "spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:

<size> == 8  
  <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16  
  <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32  
  <index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn>  
  Is the general-purpose base register, encoded in the "Rn" field.

<align>  
  Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and the encoding depends on <size>:

<size> == 8  
  Encoded in the "index_align<0>" field as 0.

<size> == 16  
  Encoded in the "index_align<0>" field as 0.

<size> == 32  
  Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8  
  <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16  
  <align> is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32  
  <align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm>  
  Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.
if \texttt{ConditionPassed}() then
\begin{verbatim}
  EncodingSpecificOperations(); \texttt{CheckAdvSIMDEnabled}();
  address = \texttt{R}[n];  iswrite = FALSE;
  - = \texttt{AArch32.CheckAlignment}(address, alignment, \texttt{AccType_VEC}, iswrite);
  \texttt{Elem}[\texttt{D}[d], index] = \texttt{MemU}[\texttt{address}, ebytes];
  \texttt{Elem}[\texttt{D}[d2], index] = \texttt{MemU}[\texttt{address}+\texttt{ebytes}, ebytes];
  \texttt{Elem}[\texttt{D}[d3], index] = \texttt{MemU}[\texttt{address}+2*\texttt{ebytes}, ebytes];
  \texttt{Elem}[\texttt{D}[d4], index] = \texttt{MemU}[\texttt{address}+3*\texttt{ebytes}, ebytes];
if \texttt{wback} then
  if \texttt{register_index} then
    \texttt{R}[n] = \texttt{R}[n] + \texttt{R}[m];
  else
    \texttt{R}[n] = \texttt{R}[n] + 4*\texttt{ebytes};
\end{verbatim}
VLD4 (single 4-element structure to all lanes)

Load single 4-element structure and replicate to all lanes of four registers loads one 4-element structure from memory into all lanes of four registers. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>D</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Vd</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>T</th>
<th>a</th>
<th></th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Offset (Rm == 1111)

```c
VLD4{<c>}{<q>}.<size> <list>, [<<Rn>{:<align}>]
```

Post-indexed (Rm == 1101)

```c
VLD4{<c>}{<q>}.<size> <list>, [<<Rn>{:<align}>]!
```

Post-indexed (Rm != 11x1)

```c
VLD4{<c>}{<q>}.<size> <list>, [<<Rn>{:<align}>],<Rm>
```

if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
  ebytes = 4;  alignment = 16;
else
  ebytes = 1 << UInt(size);
  if size == '10' then
    alignment = if a == '0' then 1 else 8;
  else
    alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

|   |   |   |   |   |   |   |   |   |   |   |   | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | D | 1 | 0 | Rn | Vd | 1 | 1 | 1 | 1 | size | T | a |   | Rm |
Offset (Rm == 1111)

VLD4\{<c>\}<q>.<size> <list>, [\{<Rn>:<align>\}]

Post-indexed (Rm == 1101)

VLD4\{<c>\}<q>.<size> <list>, [\{<Rn>:<align>\}!]

Post-indexed (Rm != 11x1)

VLD4\{<c>\}<q>.<size> <list>, [\{<Rn>:<align>\}], <Rm>

if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
  ebytes = 4;  alignment = 16;
else
  ebytes = 1 << UInt(size);
  if size == '10' then
    alignment = if a == '0' then 1 else 8;
  else
    alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD4 (single 4-element structure to all lanes).

Assembler Symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<size>\) Is the data size, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>1x</td>
<td>32</td>
</tr>
</tbody>
</table>

\(<list>\) Is a list containing the 64-bit names of four SIMD&FP registers.
The list must be one of:

{ \{ <Dd>[], <Dd+1>[], <Dd+2>[], <Dd+3>[] \} } 
Single-spaced registers, encoded in the “T” field as 0.

{ \{ <Dd>[], <Dd+2>[], <Dd+4>[], <Dd+6>[] \} } 
Double-spaced registers, encoded in the “T” field as 1.

The register <Dd> is encoded in the “D:Vd” field.

\(<Rn>\) Is the general-purpose base register, encoded in the “Rn” field.

\(<align>\) Is the optional alignment.
Whenever \(<align>\) is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the “a” field as 0.
Whenever $\langle\text{align}\rangle$ is present, the permitted values and encoding depend on $\langle\text{size}\rangle$:

$\langle\text{size}\rangle == 8$

$\langle\text{align}\rangle$ is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

$\langle\text{size}\rangle == 16$

$\langle\text{align}\rangle$ is 64, meaning 64-bit alignment, encoded in the "a" field as 1.

$\langle\text{size}\rangle == 32$

$\langle\text{align}\rangle$ can be 64 or 128. 64-bit alignment is encoded in the "a:size<0>" field as 0b10, and 128-bit alignment is encoded in the "a:size<0>" field as 0b11.

$:\$ is the preferred separator before the $\langle\text{align}\rangle$ value, but the alignment can be specified as @$\langle\text{align}\rangle$, see Advanced SIMD addressing mode.

$\langle\text{Rm}\rangle$  
Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes]);
    D[d4] = Replicate(MemU[address+3*ebytes,ebytes]);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```

Internal version only: isa v01_19, pseudocode v2020-09 xml, sve v2020-09 rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VLD4 (multiple 4-element structures)

Load multiple 4-element structures to four registers loads multiple 4-element structures from memory into four registers, with de-interleaving. For more information, see Element and structure load/store instructions. Every element of each register is loaded. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

itype

Offset (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

case itype of
  when '0000'
      inc = 1;
  when '0001'
      inc = 2;
  otherwise
      SEE "Related encodings";
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | D | 1 | 0 | Rn | Vd | 0 | 0 | 0 | x | size | align | Rm |

itype
Offset (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLD4 (multiple 4-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> Is the data size, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<list> Is a list containing the 64-bit names of the SIMD&FP registers. The list must be one of:

\{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> \}

Single-spaced registers, encoded in the "itype" field as 0b0000.

\{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> \}

Double-spaced registers, encoded in the "itype" field as 0b0001.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.
<align> Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64  
64-bit alignment, encoded in the "align" field as 0b01.

128  
128-bit alignment, encoded in the "align" field as 0b10.

256  
256-bit alignment, encoded in the "align" field as 0b11.

; is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  address = R[n]; iswrite = FALSE;
  - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
  for e = 0 to elements-1
    Elem[D[d], e] = MemU[address,ebytes];
    Elem[D[d2], e] = MemU[address+ebytes,ebytes];
    Elem[D[d3], e] = MemU[address+2*ebytes,ebytes];
    Elem[D[d4], e] = MemU[address+3*ebytes,ebytes];
    address = address + 4*ebytes;
  if wback then
    if register_index then
      R[n] = R[n] + R[m];
    else
      R[n] = R[n] + 32;
Load Multiple SIMD&FP registers loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the alias VPOP.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

\[
\begin{array}{ccccccccccc}
! & 1 & 1 & 0 & P & U & D & W & 1 & Rn & Vd & 1 & 0 & 1 & 1 & \text{imm8}<7:1> & 0 \\
& & & & & & & & & & & & & & & \text{imm8}<0> & \\
\end{array}
\]

Decrement Before (P == 1 && U == 0 && W == 1)

VLDMDB\{<c>\}{<q>}{.<size>} \langleRn\rangle!, \langledreglist\rangle

Increment After (P == 0 && U == 1)

VLDM\{<c>\}{<q>}{.<size>} \langleRn\rangle{}, \langledreglist\rangle

VLDMIA\{<c>\}{<q>}{.<size>} \langleRn\rangle{}, \langledreglist\rangle

if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \( \text{regs} == 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If \( \text{regs} > 16 \) || (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------|-----------------------------------|-----------------------------------|-----------------------------------|-----------------------------------|-----------------------------------|-----------------------------------|
| != 1111 1 1 0 P U D W 1 Rn Vd 1 0 1 0 imm8 |
| cond imm8<0> |

VLDM, VLDMDB, VLDMIA
Decrement Before \((P == 1 && U == 0 && W == 1)\)

\[
\text{VLDMDB}\{<c}\{<q>\}.<\text{size}>\ <\text{Rn}>!, <\text{dreglist}>
\]

Increment After \((P == 0 && U == 1)\)

\[
\text{VLDM}\{<c}\{<q>\}.<\text{size}>\ <\text{Rn}>!, <\text{dreglist}>
\]

\[
\text{VLDMIA}\{<c}\{<q>\}.<\text{size}>\ <\text{Rn}>!, <\text{dreglist}>
\]

if \(P == '0' \&\& U == '0' \&\& W == '0'\) then SEE "Related encodings";
if \(P == '1' \&\& W == '0'\) then SEE "VLDR";
if \(P == U \&\& W == '1'\) then UNDEFINED;

// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
\[
\text{single_regs} = \text{TRUE}; \ add = (U == '1'); \ \text{wback} = (W == '1'); \ d = \text{UInt}(Vd:D); \ n = \text{UInt}(\text{Rn});
\]
\[
\text{imm32} = \text{ZeroExtend}(\text{imm8:'00', 32}); \ \text{regs} = \text{UInt}(\text{imm8});
\]
if \(n == 15 \&\& \text{wback} \mid\mid \text{CurrentInstrSet}() != \text{InstrSet_A32}\) then UNPREDICTABLE;
if \(\text{regs} == 0 \mid\mid (d+\text{regs}) > 32\) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(\text{regs} == 0\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If \((d+\text{regs}) > 32\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>D</td>
<td>W</td>
<td>1</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
</tr>
</tbody>
</table>

Decrement Before \((P == 1 && U == 0 && W == 1)\)

\[
\text{VLDMDB}\{<c}\{<q>\}.<\text{size}>\ <\text{Rn}>!, <\text{dreglist}>
\]

Increment After \((P == 0 && U == 1)\)

\[
\text{VLDM}\{<c}\{<q>\}.<\text{size}>\ <\text{Rn}>!, <\text{dreglist}>
\]

\[
\text{VLDMIA}\{<c}\{<q>\}.<\text{size}>\ <\text{Rn}>!, <\text{dreglist}>
\]

if \(P == '0' \&\& U == '0' \&\& W == '0'\) then SEE "Related encodings";
if \(P == '1' \&\& W == '0'\) then SEE "VLDR";
if \(P == U \&\& W == '1'\) then UNDEFINED;

// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
\[
\text{single_regs} = \text{TRUE}; \ add = (U == '1'); \ \text{wback} = (W == '1'); \ d = \text{UInt}(Vd:D); \ n = \text{UInt}(\text{Rn});
\]
\[
\text{imm32} = \text{ZeroExtend}(\text{imm8:'00', 32}); \ \text{regs} = \text{UInt}(\text{imm8});
\]
if \(n == 15 \&\& \text{wback} \mid\mid \text{CurrentInstrSet}() != \text{InstrSet_A32}\) then UNPREDICTABLE;
if \(\text{regs} == 0 \mid\mid (d+\text{regs}) > 32\) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(\text{regs} == 0\), then one of the following behaviors must occur:
The instruction is UNDEFINED.
The instruction executes as NOP.
The instruction operates as a VLDM with the same addressing mode but loads no registers.

If \( \text{regs} > 16 \) \( \| \) \( \text{(d+regs)} > 32 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

If \( \text{regs} = 0 \) \( \| \) \( \text{(d+regs)} > 32 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VLDM.

Related encodings: See Advanced SIMD and floating-point 64-bit move for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move for the A32 instruction set.

Assembler Symbols

- \(<c>\) See Standard assembler syntax fields.
- \(<q>\) See Standard assembler syntax fields.
- \(<\text{size}>\) An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
- \(<\text{Rn}>\) Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.
! Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
<sreglist> Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
<dreglist> Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPOP</td>
<td>P == '0' &amp;&amp; U == '1' &amp;&amp; W == '1' &amp;&amp; Rn == '1101'</td>
</tr>
</tbody>
</table>

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4]; address = address+4;
        else
            word1 = MemA[address,4]; word2 = MemA[address+4,4]; address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;
        if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
    end;
end;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VLDR (immediate)

Load SIMD&FP register (immediate) loads a single register from the Advanced SIMD and floating-point register file, using an address from a general-purpose register, with an optional offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| cond | Rn
|------|---
| != 1111 | 1 1 0 1 1 0 |
| Rn | 1 0 | size | imm8

Half-precision scalar (size == 01)  
(Armv8.2)

VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/−}<imm}>]

Single-precision scalar (size == 10)

VLDR{<c>}{<q>{.32} <Sd>, [<Rn> {, #{+/−}<imm}>]}

Double-precision scalar (size == 11)

VLDR{<c>}{<q>{.64} <Dd>, [<Rn> {, #{+/−}<imm}>]}

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);

if size == '01' && cond != '1110', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

| cond | Rn
|------|---
| != 1111 | 1 1 0 1 1 0 |
| Rn | 1 0 | size | imm8

CONstrained UNPREDICTABLE behavior
Half-precision scalar (size == 01)

(Armv8.2)

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #/+/-}<imm>]
```

Single-precision scalar (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #/+/-}<imm>]
```

Double-precision scalar (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #/+/-}<imm>]
```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
n = UInt(Rn);

CONSTRUED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `.64` Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `.32` Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
- `/+` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th><code>U</code></th>
<th><code>/+</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

- `<imm>` For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as `<imm>/4`.
  For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as `<imm>/2`.

VLDR (immediate)
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16
            S[d] = Zeros(16) : MemA[address,2];
        when 32
            S[d] = MemA[address,4];
        when 64
            word1 = MemA[address,4]; word2 = MemA[address+4,4];
            // Combine the word-aligned words in the correct order for current endianness.
            D[d] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VLDR (literal)

Load SIMD&FP register (literal) loads a single register from the Advanced SIMD and floating-point register file, using an address from the PC value and an immediate offset. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>!= 1111</th>
<th>1 1 0 1</th>
<th>U</th>
<th>D</th>
<th>0</th>
<th>1 1 1 1 1</th>
<th>Vd</th>
<th>1 0</th>
<th>size</th>
<th>imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision scalar (size == 01)
(Armv8.2)

VLDR{<c>}{<q>}.16 <Sd>, <label>
VLDR{<c>}{<q>}.16 <Sd>, [PC, #<+/-><imm>]

Single-precision scalar (size == 10)

VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #<+/-><imm>]

Double-precision scalar (size == 11)

VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #<+/-><imm>]

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
n = UInt(Rn);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

| 1 1 1 0 1 1 0 | 1 | U | D | 0 | 1 1 1 1 1 | Vd | 1 0 | size | imm8 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Rn |
Half-precision scalar (size == 01)  
(Armv8.2)

VLDR{<c}<{q}>/.16 <Sd>, <label>
VLDR{<c}<{q}>/.16 <Sd>, [PC, #{+/−}<imm>]

Single-precision scalar (size == 10)

VLDR{<c}<{q}>/.32 <Sd>, <label>
VLDR{<c}<{q}>/.32 <Sd>, [PC, #{+/−}<imm>]

Double-precision scalar (size == 11)

VLDR{<c}<{q}>/.64 <Dd>, <label>
VLDR{<c}<{q}>/.64 <Dd>, [PC, #{+/−}<imm>]

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
  when '01'  d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
n = UInt(Rn);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

.64 Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

.32 Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<label> The label of the literal data item to be loaded.

For the single-precision scalar or double-precision scalar variants: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020.

For the half-precision scalar variant: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 2 in the range -510 to 510.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.
If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

+/− Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

VLDR (literal)
For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the “imm8” field as <imm>/4.

For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the “imm8” field as <imm>/2.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16
            S[d] = Zeros(16) : MemA[address,2];
        when 32
            S[d] = MemA[address,4];
        when 64
            word1 = MemA[address,4]; word2 = MemA[address+4,4];
            // Combine the word-aligned words in the correct order for current endianness.
            D[d] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;
```
VMAX (floating-point)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

64-bit SIMD vector (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

if sz == '1' && !HaveFP16Ext() then UNDEFINED;

maximum = (op == '0');

case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;

  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

if sz == '1' && !HaveFP16Ext() then UNDEFINED;

if sz == '1' && InITBlock() then UNPREDICTABLE;

maximum = (op == '0');

case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;

  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRANED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler Symbols**

- `<c>`: For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  For encoding T1: see *Standard assembler syntax fields*.
- `<q>`: See *Standard assembler syntax fields*.
- `<dt>`: Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as `<Qd>*2.
- `<Qn>`: Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.
- `<Qm>`: Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.
- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dn>`: Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- `<Dm>`: Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Floating-point maximum and minimum**

- $\max(0.0, -0.0) = +0.0$
- If any input is a NaN, the corresponding result element is the default NaN.

**Operation**

```markdown
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**VMAX (integer)**

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements. Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see **Enabling Advanced SIMD and floating-point support**.

It has encodings from the following instruction sets: A32 (**A1** ) and T32 (**T1** ).

**A1**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 U 0 D size Vn Vd 0 1 1 0 N Q M 0 Vm
```

64-bit SIMD vector (Q == 0)

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

**T1**

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 U 1 1 1 1 0 D size Vn Vd 0 1 1 0 N Q M 0 Vm
```

64-bit SIMD vector (Q == 0)

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

**Assembler Symbols**

- `<c>` For encoding A1: see **Standard assembler syntax fields**. This encoding must be unconditional.
  For encoding T1: see **Standard assembler syntax fields**.

- `<q>` See **Standard assembler syntax fields**.
Is the data type for the elements of the operands, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r], e, esize], unsigned);
            op2 = Int(Elem[D[m+r], e, esize], unsigned);
            result = if maximum then Max(op1, op2) else Min(op1, op2);
            Elem[D[d+r], e, esize] = result<esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

This instruction determines the floating-point maximum number. It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point VMAX. This instruction is not conditional.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

64-bit SIMD vector (Q == 0)

VMAXNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMAXNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

if Q == '1' & & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & & !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

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

Half-precision scalar (size == 01)  
(Armv8.2)

VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Single-precision scalar (size == 10)

VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Double-precision scalar (size == 11)

VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

if size == '00' || (size == '01' & & !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1
64-bit SIMD vector (Q == 0)

\[ VMAXNM\{\langle q\rangle\}.<dt> \langle Dd\rangle, \langle Dn\rangle, \langle Dm\rangle \]

128-bit SIMD vector (Q == 1)

\[ VMAXNM\{\langle q\rangle\}.<dt> \langle Qd\rangle, \langle Qn\rangle, \langle Qm\rangle \]

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

\[ \text{size} \quad \text{op} \]

Half-precision scalar (size == 01)
(Armv8.2)

\[ VMAXNM\{\langle q\rangle\}.F16 \langle Sd\rangle, \langle Sn\rangle, \langle Sm\rangle // (Not permitted in IT block) \]

Single-precision scalar (size == 10)

\[ VMAXNM\{\langle q\rangle\}.F32 \langle Sd\rangle, \langle Sn\rangle, \langle Sm\rangle // (Not permitted in IT block) \]

Double-precision scalar (size == 11)

\[ VMAXNM\{\langle q\rangle\}.F64 \langle Dd\rangle, \langle Dn\rangle, \langle Dm\rangle // (Not permitted in IT block) \]

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the **CONstrained UNpredictable** behavior of this instruction, see *Architectural Constraints on UNpredictable behaviors*.

**Assembler Symbols**

| <q> | See *Standard assembler syntax fields*.
| <dt> | Is the data type for the elements of the vectors, encoded in “sz”: |
| sz | <dt> |
| 0 | F32 |
| 1 | F16 |

| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
| <Qn> | Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2. |
| <Qm> | Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2. |
| <Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field. |
| <Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field. |

**Operation**

```plaintext
EncodingSpecificOperations(); CheckAdvSIMDorVFPEnabled(TRUE, advsimd);

if advsimd then // Advanced SIMD instruction
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
      if maximum then
        Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
      else
        Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
    else
      // VFP instruction
      case esize of
        when 16
          if maximum then
            S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
          else
            S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        when 32
          if maximum then
            S[d] = FPMaxNum(S[n], S[m], FPSCR[]);
          else
            S[d] = FPMinNum(S[n], S[m], FPSCR[]);
        when 64
          if maximum then
            D[d] = FPMaxNum(D[n], D[m], FPSCR[]);
          else
            D[d] = FPMinNum(D[n], D[m], FPSCR[]);
Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
```

VMAXNM
**VMIN (floating-point)**

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector. The operand vector elements are floating-point numbers. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

**64-bit SIMD vector (Q == 0)**

VMIN{<c>{<q>}.<dt> {<Dd>, }<Dn>, <Dm}.

**128-bit SIMD vector (Q == 1)**

VMIN{<c>{<q>}.<dt> {<Qd>, }<Qn>, <Qm}.

- If Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
- If sz == '1' && !HaveFP16Ext() then UNDEFINED;
- Maximum = (op == '0');
- Case sz of
  - when '0' esize = 32; elements = 2;
  - when '1' esize = 16; elements = 4;
  - d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---:| :---: |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | D | 1 | sz | Vn | Vd | 1 | 1 | 1 | 1 | N | Q | M | 0 | Vm |

**64-bit SIMD vector (Q == 0)**

VMIN{<c>{<q>}.<dt> {<Dd>, }<Dn>, <Dm}.

**128-bit SIMD vector (Q == 1)**

VMIN{<c>{<q>}.<dt> {<Qd>, }<Qn>, <Qm}.

- If Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
- If sz == '1' && !HaveFP16Ext() then UNDEFINED;
- If sz == '1' && InITBlock() then UNPREDICTABLE;
- Maximum = (op == '0');
- Case sz of
  - when '0' esize = 32; elements = 2;
  - when '1' esize = 16; elements = 4;
  - d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**CONSTRAINED UNPREDICTABLE behavior**

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler Symbols**

- `<c>`: For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  For encoding T1: see *Standard assembler syntax fields*.
- `<q>`: See *Standard assembler syntax fields*.
- `<dt>`: Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- `<Qn>`: Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.
- `<Qm>`: Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.
- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dn>`: Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- `<Dm>`: Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Floating-point minimum**

- \( \min(+0.0, -0.0) = -0.0 \)
- If any input is a NaN, the corresponding result element is the default NaN.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**VMIN (integer)**

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

**A1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 1  | U  | 0  | D  | size | Vn | Vd | 0  | 1  | 1  | 0  | N  | Q  | M  | 1  | Vm | op |

**64-bit SIMD vector (Q == 0)**

\[
\text{VMIN} \{<c>\} \{<q>\}.<dt> \{<Dd>, }<Dn>, <Dm>
\]

**128-bit SIMD vector (Q == 1)**

\[
\text{VMIN} \{<c>\} \{<q>\}.<dt> \{<Qd>, }<Qn>, <Qm>
\]

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | U  | 1  | 1  | 1  | 1  | 0  | D  | size | Vn | Vd | 0  | 1  | 1  | 0  | N  | Q  | M  | 1  | Vm | op |

**64-bit SIMD vector (Q == 0)**

\[
\text{VMIN} \{<c>\} \{<q>\}.<dt> \{<Dd>, }<Dn>, <Dm>
\]

**128-bit SIMD vector (Q == 1)**

\[
\text{VMIN} \{<c>\} \{<q>\}.<dt> \{<Qd>, }<Qn>, <Qm>
\]

**Assembler Symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  
  For encoding T1: see *Standard assembler syntax fields*.

- `<q>` See *Standard assembler syntax fields*.
Is the data type for the elements of the operands, encoded in "U:size":

<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction determines the floating point minimum number. It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point VMIN. This instruction is not conditional.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

**A1**

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

**64-bit SIMD vector (Q == 0)**

VMINNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VMINNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**A2**

| 31 30 29 28 27 26 25 24 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 1 1 1 0 1 | D | 0 0 | Vn | Vd | 1 0 != 00 N 1 M 0 | Vm |

**Half-precision scalar (size == 01)**

(Armv8.2)

VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

**Single-precision scalar (size == 10)**

VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

**Double-precision scalar (size == 11)**

VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**T1**
64-bit SIMD vector \((Q == 0)\)

\[
\text{VMINNM}\{<q>\}.<dt> <Dd>, <Dn>, <Dm>
\]

128-bit SIMD vector \((Q == 1)\)

\[
\text{VMINNM}\{<q>\}.<dt> <Qd>, <Qn>, <Qm>
\]

If \text{InITBlock}() then UNPREDICTABLE;
if \(Q == '1' \&\& (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1')\) then UNDEFINED;
if \(sz == '1' \&\& !\text{HaveFP16Ext}()\) then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = \text{UInt}(D:Vd); n = \text{UInt}(N:Vn); m = \text{UInt}(M:Vm); regs = if Q == '0' then 1 else 2;

\textbf{CONSTRAINED UNPREDICTABLE behavior}

If \text{InITBlock}(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

\textbf{T2}

Half-precision scalar \((\text{size} == 01)\)
\((\text{Armv8.2})\)

\[
\text{VMINNM}\{<q>\}.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
\]

Single-precision scalar \((\text{size} == 10)\)

\[
\text{VMINNM}\{<q>\}.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
\]

Double-precision scalar \((\text{size} == 11)\)

\[
\text{VMINNM}\{<q>\}.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
\]

if \text{InITBlock}() then UNPREDICTABLE;
if size == '00' || (size == '01' \&\& !\text{HaveFP16Ext}()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
case size of
  when '01' esize = 16; d = \text{UInt}(Vd:D); n = \text{UInt}(Vn:N); m = \text{UInt}(Vm:M);
  when '10' esize = 32; d = \text{UInt}(Vd:D); n = \text{UInt}(Vn:N); m = \text{UInt}(Vm:M);
  when '11' esize = 64; d = \text{UInt}(D:Vd); n = \text{UInt}(N:Vn); m = \text{UInt}(M:Vm);

\textbf{CONSTRAINED UNPREDICTABLE behavior}

If \text{InITBlock}(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONstrained UNpredictable behavior of this instruction, see Architectural Constraints on UNpredictable behaviors.

Assembler Symbols

<q>
See Standard assembler syntax fields.

<dt>
Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qn>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd>
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

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

<Sm>
Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then // Advanced SIMD instruction
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
      if maximum then
        Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
      else
        Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
    else
      // VFP instruction
      case esize of
        when 16
          if maximum then
            S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR());
          else
            S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR());
        when 32
          if maximum then
            S[d] = FPMaxNum(S[n], S[m], FPSCR());
          else
            S[d] = FPMinNum(S[n], S[m], FPSCR());
        when 64
          if maximum then
            D[d] = FPMaxNum(D[n], D[m], FPSCR());
          else
            D[d] = FPMinNum(D[n], D[m], FPSCR());

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VMLA (floating-point)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

### A1

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

### A2

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

if Q == '1' & (Vd<0> == '0' || Vn<0> == '0') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; add = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M); regs = if Q == '0' then 1 else 2;

### A3

32-bit SIMD vector (Q == 11)

VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Half-precision scalar (size == 01)
(Armv8.2)

VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond !='1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
CONSTRANDED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

op

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; add = (op == '0');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRANDED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
</tbody>
</table>

op

VMLA (floating-point)  Page 950
Half-precision scalar (size == 01)
(Armv8.2)

VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding A2, T1 and T2: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the vectors, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, StandardFPSCRValue());
        else // VFP instruction
            case esize of
                when 16
                    addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]) else FPNeg(FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]));
                    S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR[]);
                when 32
                    addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else FPNeg(FPMul(S[n], S[m], FPSCR[]));
                    S[d] = FPAdd(S[d], addend32, FPSCR[]);
                when 64
                    addend64 = if add then FPMul(Q[n], Q[m], FPSCR[]) else FPNeg(FPMul(Q[n], Q[m], FPSCR[]));
                    D[d] = FPAdd(D[d], addend64, FPSCR[]);
VMLA (integer)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and adds the products to the corresponding elements of the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // (Encoding T1/A1, encoded as Q = 0)

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // (Encoding T1/A1, encoded as Q = 1)

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0'); long_destination = FALSE;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 1 1 0 0 0 1 0 0 1</td>
</tr>
<tr>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // (Encoding T1/A1, encoded as Q = 0)

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // (Encoding T1/A1, encoded as Q = 1)

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0'); long_destination = FALSE;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

?type> The data type for the elements of the operands. It must be one of:
S  Optional in encoding T1/A1. Encoded as $U = 0$ in encoding T2/A2.

U  Optional in encoding T1/A1. Encoded as $U = 1$ in encoding T2/A2.

I  Available only in encoding T1/A1.

<size> The data size for the elements of the operands. It must be one of:

8  Encoded as size = 0b00.

16  Encoded as size = 0b01.

32  Encoded as size = 0b10.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as $<Q_d>*2$.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as $<Q_n>*2$.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as $<Q_m>*2$.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din][n+r],e,esize],unsigned) * Int(Elem[Din][m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q][d>>1],e,2*esize] = Elem[Qin][d>>1],e,2*esize] + addend;
            else
                Elem[D][d+r],e,esize] = Elem[Din][d+r],e,esize] + addend;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Multiply Accumulate multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 1 1 0 0 1 | Q | 1 | D | != 11 | Vn | Vd | 0 | 0 | 0 | F | N | 1 | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<8> == '1' || Vn<8> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 1 1 | Q | 1 | 1 | 1 | 1 | D | != 11 | Vn | Vd | 0 | 0 | 0 | F | N | 1 | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<8> == '1' || Vn<8> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>
For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<dt>
Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

<table>
<thead>
<tr>
<th>F</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>I16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>I32</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd>
Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd>
Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<DM[x]>
Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2;0>", and x is encoded in "M:Vm<3>". If <dt> is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  op2 = Elem[Din[m],index,esize]; op2val = Int(op2, unsigned);
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[Din[n+r],e,esize]; op1val = Int(op1, unsigned);
      if floating_point then
        if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
        else
          addend = if add then op1val*op2val else -op1val*op2val;
          if long_destination then
            Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
          else
            Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
  }

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Multiply Accumulate Long multiplies corresponding elements in two vectors, and add the products to the corresponding element of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 | U | 1 | D | != 11 | Vn | Vd | 1 0 0 0 | N | 0 | M | 0 | Vm
```

size | op
---|---

A1

```
VMLAL{<c>}{<q>}.{<type>}{<size>}{<Qd>}, {<Dn>}, {<Dm>} // (Encoding T2/A2)

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0'); long_destination = TRUE; unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
```

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 | U | 1 1 1 1 1 | D | != 11 | Vn | Vd | 1 0 0 0 | N | 0 | M | 0 | Vm
```

size | op
---|---

T1

```
VMLAL{<c>}{<q>}.{<type>}{<size>}{<Qd>}, {<Dn>}, {<Dm>} // (Encoding T2/A2)

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0'); long_destination = TRUE; unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

| <c> | For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1: see Standard assembler syntax fields. |
| <q> | See Standard assembler syntax fields. |
| <type> | The data type for the elements of the operands. It must be one of: S Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2. U Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2. I Available only in encoding T1/A1. |
The data size for the elements of the operands. It must be one of:

- **8**
  - Encoded as size = 0b00.
- **16**
  - Encoded as size = 0b01.
- **32**
  - Encoded as size = 0b10.

**<Qd>**  
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

**<Dn>**  
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

**<Dm>**  
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Multiply Accumulate Long multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#))

### A1

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

size | op

#### A1

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

if size == '11' then SEE “Related encodings”;
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); add = (op == '0'); floating_point = FALSE; long_destination = TRUE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

#### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
</tr>
</tbody>
</table>

size | op

#### T1

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

if size == '11' then SEE “Related encodings”;
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); add = (op == '0'); floating_point = FALSE; long_destination = TRUE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

### Assembler Symbols

- **<c>**
  - For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.
  - For encoding T1: see [Standard assembler syntax fields](#).

- **<q>**
  - See [Standard assembler syntax fields](#).

- **<dt>**
  - Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U</th>
<th>dt</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>U16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>U32</td>
</tr>
</tbody>
</table>
<Qd>  Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]>  Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3">. If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
      if floating_point then
        fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
        Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
      else
        addend = if add then op1val*op2val else -op1val*op2val;
        if long_destination then
          Elem[Q[d>>1],e,2*esize] = Elem[Din[d>>1],e,2*esize] + addend;
        else
          Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Multiply Subtract multiplies corresponding elements in two vectors, subtracts the products from corresponding elements of the destination vector, and places the results in the destination vector.

Arm recommends that software does not use the VMLS instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes. Depending on settings in the **CPACR, NSACR, HCPRTR**, and **FPEXC** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see **Enabling Advanced SIMD and floating-point support**.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

**A1**

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

64-bit SIMD vector (Q == 0)

VMLS{<c>}{<q>}.{dt} <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLS{<c>}{<q>}.{dt} <Qd>, <Qn>, <Qm>

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; add = (op == '0');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**A2**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| != | 1111 | 1  | 1  | 1  | 0  | 1  | 0  | D  | 0  | 0  | Vn |   |   | Vd | 1  | 0  |   | size | N  | 1  | M  | 0  | Vm |

cond op
Half-precision scalar (size == 01)
(Armv8.2)

VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>D</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>1</td>
</tr>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VMLS{<c>}{<q>}.<dt> <Od>, <On>, <Om>

128-bit SIMD vector (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && !InITBlock() then UNPREDICTABLE;
advsimd = TRUE; add = (op == '0');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 64; elements = 4;
  d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2
Half-precision scalar (size == 01) (Armv8.2)

VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<ct> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding A2, T1 and T2: see Standard assembler syntax fields.
<qt> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qdt> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qnt> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qmt> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Ddt> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dnt> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dmt> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Sdt> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Snt> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Smt> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[r+1],e,esize], addend, StandardFPSCRValue());
    else // VFP instruction
        case esize of
            when 16
                addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]) else FPNeg(FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]));
                S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR[]);
            when 32
                addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else FPNeg(FPMul(S[n], S[m], FPSCR[]));
                S[d] = FPAdd(S[d], addend32, FPSCR[]);
            when 64
                addend64 = if add then FPMul(D[n], D[m], FPSCR[]) else FPNeg(FPMul(D[n], D[m], FPSCR[]));
                D[d] = FPAdd(D[d], addend64, FPSCR[]);
        end case
end if
VMLS (integer)

Vector Multiply Subtract multiplies corresponding elements in two vectors, and subtracts the products from the corresponding elements of the destination vector. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

64-bit SIMD vector (Q == 0)

VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // (Encoding T1/A1, encoded as Q = 0)

128-bit SIMD vector (Q == 1)

VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // (Encoding T1/A1, encoded as Q = 1)

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>1 1 1 1 1 1 1 1 0</th>
<th>D size</th>
<th>Vn</th>
<th>Vd</th>
<th>1 0 0 1</th>
<th>N</th>
<th>Q</th>
<th>M</th>
<th>0</th>
<th>Vm</th>
</tr>
</thead>
</table>

64-bit SIMD vector (Q == 0)

VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // (Encoding T1/A1, encoded as Q = 0)

128-bit SIMD vector (Q == 1)

VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // (Encoding T1/A1, encoded as Q = 1)

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<type> The data type for the elements of the operands. It must be one of:
**S**  
Optional in encoding T1/A1. Encoded as $U = 0$ in encoding T2/A2.

**U**  
Optional in encoding T1/A1. Encoded as $U = 1$ in encoding T2/A2.

**I**  
Available only in encoding T1/A1.

<size>  
The data size for the elements of the operands. It must be one of:

- **8**  
  Encoded as $\text{size} = 0b00$.

- **16**  
  Encoded as $\text{size} = 0b01$.

- **32**  
  Encoded as $\text{size} = 0b10$.

<Qd>  
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as $<Qd> \times 2$.

<Qn>  
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as $<Qn> \times 2$.

<Qm>  
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as $<Qm> \times 2$.

<Dd>  
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>  
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>  
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdySIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din][n+r],e,esize],unsigned) * Int(Elem[Din][m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Multiply Subtract multiplies elements of a vector by a scalar, and either subtracts the products from corresponding elements of the destination vector.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VMLS{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>}

128-bit SIMD vector (Q == 1)

VMLS{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>}

if size == '11' then SEE "Related encodings";
if size == '00' | (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VMLS{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>}

128-bit SIMD vector (Q == 1)

VMLS{<c>{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>}

if size == '11' then SEE "Related encodings";
if size == '00' | (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:
The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

### Assembler Symbols

- `<c>`: For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
  - For encoding T1: see Standard assembler syntax fields.
- `<q>`: See Standard assembler syntax fields.
- `<dt>`: Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:
  
<table>
<thead>
<tr>
<th>F</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>I16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>I32</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

- `<Qd>`: Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field as `<Qd>="2`.
- `<Qn>`: Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as `<Qn>="2`.
- `<Dd>`: Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field.
- `<Dn>`: Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.
- `<Dm[x]>`: Is the 64-bit name of the second SIMD&FP source register holding the scalar. If `<dt>` is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If `<dt>` is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

### Operation

```java
if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
      if floating_point then
        fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
        op = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
      else
        addend = if add then op1val*op2val else -op1val*op2val;
        if long_destination then
          Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
        else
          Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Multiply Subtract Long multiplies corresponding elements in two vectors, and subtract the products from the corresponding elements of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 30 29 28 27 26 25 24 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 | Vn | Vd | 1 0 1 0 | N | M | 0 | Vm |
|---|---|---|---|---|---|---|---|
| 1 1 1 0 0 1 U 1 D != 11 | op |

**A1**

VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // (Encoding T2/A2)

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | size | Vn | Vd | 1 0 1 0 | N | M | 0 | Vm |
|---|---|---|---|---|---|---|---|
| 1 1 1 U 1 1 1 1 1 1 D != 11 | op |

**T1**

VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // (Encoding T2/A2)

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

**Assembler Symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  For encoding T1: see *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<type>` The data type for the elements of the operands. It must be one of:
  - `S` Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.
  - `U` Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.
  - `I` Available only in encoding T1/A1.
The data size for the elements of the operands. It must be one of:

- 8
  - Encoded as size = 0b00.
- 16
  - Encoded as size = 0b01.
- 32
  - Encoded as size = 0b10.

**<Qd>**

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

**<Dn>**

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

**<Dm>**

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Multiply Subtract Long multiplies elements of a vector by a scalar, and subtracts the products from corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
1 1 1 1 0 0 1 | U | 1 | D | != 11 | Vn | Vd | 0 | 1 | 0 | N | 1 | M | 0 | Vm

size         op
```

A1

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

if size == ‘11’ then SEE “Related encodings”;
if size == ‘00’ || Vd<0> == ‘1’ then UNDEFINED;
unsigned = (U == ‘1’);  add = (op == ‘0’);  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == ‘01’ then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == ‘10’ then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

T1

```
15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
1 1 1 | U | 1 1 1 1 | 1 | D | != 11 | Vn | Vd | 0 | 1 | 0 | N | 1 | M | 0 | Vm

size         op
```

T1

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

if size == ‘11’ then SEE “Related encodings”;
if size == ‘00’ || Vd<0> == ‘1’ then UNDEFINED;
unsigned = (U == ‘1’);  add = (op == ‘0’);  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == ‘01’ then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == ‘10’ then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize]; op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize]; op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Din[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
    endfor
endfor

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

BFloat16 floating-point matrix multiply-accumulate. This instruction multiplies the 2x4 matrix of BF16 values in the first 128-bit source vector by the 4x2 BF16 matrix in the second 128-bit source vector. The resulting 2x2 single-precision matrix product is then added destructively to the 2x2 single-precision matrix in the 128-bit destination vector. This is equivalent to performing a 4-way dot product per destination element. The instruction does not update the FPSCR exception status.

Arm expects that the VMMLA instruction will deliver a peak BF16 multiply throughput that is at least as high as can be achieved using two VDOT instructions, with a goal that it should have significantly higher throughput.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1
(Armv8.6)

```
31 30 29 28 27 26 25 24 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 1 0 0 0 D 0 0 Vn Vd 1 1 0 0 N 1 M 0 Vm
```

A1
VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = 2;

T1
(Armv8.6)

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 0 0 0 D 0 0 Vn Vd 1 1 0 0 N 1 M 0 Vm
```

T1
VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = 2;

Assembler Symbols

<q> See Standard assembler syntax fields.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation

CheckAdvSIMDEnabled();

bits(128) op1 = Q[n>>1];
bits(128) op2 = Q[m>>1];
bits(128) acc = Q[d>>1];

Q[d>>1] = BFMatMulAdd(acc, op1, op2);
VMOV (between two general-purpose registers and a doubleword floating-point register)

Copy two general-purpose registers to or from a SIMD&FP register copies two words from two general-purpose registers into a doubleword register in the Advanced SIMD and floating-point register file, or from a doubleword register in the Advanced SIMD and floating-point register file to two general-purpose registers. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------------------------------|-------------------------------------|
| != 1111 | 1 1 0 0 0 1 0 0 1 0 0 1 1 0 | Rt2 | Rt | 1 0 1 1 0 0 | M | 1 | Vm |

From general-purpose registers (op == 0)

VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>

To general-purpose registers (op == 1)

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>

to_arm_registers = (op == '1'); t = UInt(Rt); t2 = UInt(Rt2); m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if to_arm_registers & t == t2 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If to_arm_registers && t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------------------------------|-------------------------------------|
| 1 1 1 0 1 1 0 0 0 1 0 | op | Rt2 | Rt | 1 0 1 1 0 0 | M | 1 | Vm |

From general-purpose registers (op == 0)

VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>

To general-purpose registers (op == 1)

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>

to_arm_registers = (op == '1'); t = UInt(Rt); t2 = UInt(Rt2); m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if to_arm_registers & t == t2 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If to_arm_registers && t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
The instruction executes as NOP.
The value in the destination register is UNKNOWN.

For more information about the CONstrained UNPREDICTable behavior of this instruction, see Architectural Constraints on UNPREDICTable behaviors, and particularly VMOV (between two general-purpose registers and a doubleword floating-point register).

Assembler Symbols

<Dm> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "M:Vm" field.
<Rt2> Is the second general-purpose register that <Dm>[63:32] will be transferred to or from, encoded in the "Rt2" field.
<Rt> Is the first general-purpose register that <Dm>[31:0] will be transferred to or from, encoded in the "Rt" field.
<c> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = D[m]<31:0>;
        R[t2] = D[m]<63:32>;
    else
        D[m]<31:0> = R[t];
        D[m]<63:32> = R[t2];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in the bottom 16 bits of a 32-bit SIMD&FP register to the bottom 16 bits of a general-purpose register, or the value held in the bottom 16 bits of a general-purpose register to the bottom 16 bits of a 32-bit SIMD&FP register. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1 (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 |
| != 1111 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | op | Vn | Rt | 1 | 0 | 0 | 1 | N | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
```
cond

From general-purpose register (op == 0)

```
VMOV{<c>}{<q>}.F16 <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>}.F16 <Rt>, <Sn>
```

if !HaveFP16Ext() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
to arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1 (Armv8.2)

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | op | Vn | Rt | 1 | 0 | 0 | 1 | N | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
```

From general-purpose register (op == 0)

```
VMOV{<c>}{<q>}.F16 <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>}.F16 <Rt>, <Sn>
```

if !HaveFP16Ext() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
to arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
CONSTRANDED UNPREDICTABLE behavior

If \texttt{InITBlock()}, then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as \texttt{NOP}. This means it behaves as if it fails the Condition code check.

For more information about the \texttt{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

\begin{itemize}
  \item \texttt{<Rt>} is the general-purpose register that \texttt{<Sn>} will be transferred to or from, encoded in the "Rt" field.
  \item \texttt{<Sn>} is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
  \item \texttt{<c>} see Standard assembler syntax fields.
  \item \texttt{<q>} see Standard assembler syntax fields.
\end{itemize}

Operation

\begin{verbatim}
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    if to_arm_register then
        R[t] = Zeros(16) : S[n]<15:0>;
    else
        S[n] = Zeros(16) : R[t]<15:0>;
\end{verbatim}

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Copy immediate value to a SIMD&FP register places an immediate constant into every element of the destination register. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, A3, A4, and A5) and T32 (T1, T2, T3, T4, and T5).

### A1

#### 64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.I32 <Dd>, #<imm>

#### 128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.I32 <Qd>, #<imm>

if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)"
if op == '1' && cmode != '1110' then SEE "Related encodings"
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

### A2

#### Half-precision scalar (size == 01)

(Armv8.2)

VMOV{<c>}{<q>}.F16 <Sd>, #<imm>

#### Single-precision scalar (size == 10)

VMOV{<c>}{<q>}.F32 <Sd>, #<imm>

#### Double-precision scalar (size == 11)

VMOV{<c>}{<q>}.F64 <Dd>, #<imm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
bits(16) imm16;
bits(32) imm32;
bits(64) imm64;
case size of
when '01' d = UInt(Vd:D); imm16 = VFPExpandImm(imm4H:imm4L); imm32 = Zeros(16): imm16;
when '10' d = UInt(Vd:D); imm32 = VFPExpandImm(imm4H:imm4L);
when '11' d = UInt(D:Vd); imm64 = VFPExpandImm(imm4H:imm4L); regs = 1;
**CONSTRAINED UNPREDICTABLE behavior**

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

### A3

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm3</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

cmode  op

### 64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.I16 <Dd>, #<imm>

### 128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.I16 <Qd>, #<imm>

### A4

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm3</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

cmode  op

### 64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>

### 128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>

### A5

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm3</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

cmode  op
64-bit SIMD vector (Q == 0)

VMOV<q>.I64 <D>, #imm

128-bit SIMD vector (Q == 1)

VMOV<q>.I64 <Q>, #imm

if op == '0' && cmode==0 == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode == '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector (Q == 0)

VMOV<q>.I32 <D>, #imm

128-bit SIMD vector (Q == 1)

VMOV<q>.I32 <Q>, #imm

if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode == '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

T2
Half-precision scalar (size == 01)
(Armv8.2)

VMOV{<c>}{<q>}.F16 <Sd>, #<imm>

Single-precision scalar (size == 10)

VMOV{<c>}{<q>}.F32 <Sd>, #<imm>

Double-precision scalar (size == 11)

VMOV{<c>}{<q>}.F64 <Dd>, #<imm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
bits(16) imm16;
bits(32) imm32;
bits(64) imm64;
case size of
  when '01' d = UInt(Vd:D);  imm16 = VFPExpandImm(imm4H:imm4L); imm32 = Zeros(16) : imm16;
  when '10' d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L);
  when '11' d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L);  regs = 1;

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

  • The instruction is UNDEFINED.
  • The instruction executes as if it passes the Condition code check.
  • The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T3

<table>
<thead>
<tr>
<th>1 1 1 1 0 0 0 0</th>
<th>imm3</th>
<th>Vd</th>
<th>1 0 x 0 0</th>
<th>Q</th>
<th>1</th>
<th>imm4</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmode</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.I16 <Qd>, #<imm>

if op == '0' && cmode<8> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)"
if op == '1' && cmode != '1110' then SEE Related encodings"
if Q == '1' && Vd<8> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
  d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

T4

<table>
<thead>
<tr>
<th>1 1 1 1 0 0 0 0</th>
<th>imm3</th>
<th>Vd</th>
<th>1 1 x x 0</th>
<th>Q</th>
<th>1</th>
<th>imm4</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmode</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
64-bit SIMD vector ($Q == 0$)

VMOV{$c}{q}\cdot{dt}\ <{Dd}>$, #<imm>

128-bit SIMD vector ($Q == 1$)

VMOV{$c}{q}\cdot{dt}\ <{Qd}>$, #<imm>

if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)"
if op == '1' && cmode != '1110' then SEE "Related encodings"
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

T5

```
  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1 0
  1 1 1 i 1 1 1 1 1 D 0 0 0 imm3 Vd 1 1 1 0 0 Q 1 1 imm4
```

64-bit SIMD vector ($Q == 0$)

VMOV{$c}{q}\cdot{I64} \ <{Dd}>$, #<imm>

128-bit SIMD vector ($Q == 1$)

VMOV{$c}{q}\cdot{I64} \ <{Qd}>$, #<imm>

if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)"
if op == '1' && cmode != '1110' then SEE "Related encodings"
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

- $<c>$: For encoding A1, A3, A4 and A5: see Standard assembler syntax fields. This encoding must be unconditional.
- $<q>$: For encoding A2, T1, T2, T3, T4 and T5: see Standard assembler syntax fields.
- $<dt>$: The data type, encoded in "cmode":

<table>
<thead>
<tr>
<th>cmode</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>110x</td>
<td>I32</td>
</tr>
<tr>
<td>1110</td>
<td>I8</td>
</tr>
<tr>
<td>1111</td>
<td>F32</td>
</tr>
</tbody>
</table>

- $<Qd>$: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as $<Qd>\cdot{2}$.
- $<Dd>$: Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- $<Sd>$: Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- $<imm>$: For encoding A1, A3, A4, A5, T1, T3, T4 and T5: is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of $<imm>$, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

For encoding A2 and T2: is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "imm4H:imm4L". For details of the range of constants available and the encoding of $<imm>$, see Modified immediate constants in T32 and A32 floating-point instructions.
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        \texttt{S[d]} = \texttt{imm32};
    else
        for r = 0 to regs-1
            \texttt{D[d+r]} = \texttt{imm64};

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Copy between FP registers copies the contents of one FP register to another. Depending on settings in the **CPACR**, **NSACR**, **HCPTR**, and **FPEXC** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A2](#)) and T32 ([T2](#)).

### A2

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| != 1111 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 0 | 0 | 0 | 0 | Vd | 1 | 0 | 1 | x | 0 | 1 | M | 0 | Vm |
```

- **cond**: size

**Single-precision scalar (size == 10)**

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

**Double-precision scalar (size == 11)**

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

```python
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (size == '10');  advsimd = FALSE;
if single_register then
d = UInt(Vd:D);  m = UInt(Vm:M);
else
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```

### T2

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | Vd | 1 | 0 | 1 | x | 0 | 1 | M | 0 | Vm |
```

**Assembler Symbols**

- **<c>** See *Standard assembler syntax fields*.
- **<q>** See *Standard assembler syntax fields*.
- **<Sd>** Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- **<Sm>** Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDorVFPEnabled(TRUE, advsimd);
    if single_register then
        \[D\][d] = \[D\][m];
    else
        for r = 0 to regs-1
            \[D\][d+r] = \[D\][m+r];
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Copy a general-purpose register to a vector element copies a byte, halfword, or word from a general-purpose register into an Advanced SIMD scalar.

On a Floating-point-only system, this instruction transfers one word to the upper or lower half of a double-precision floating-point register from a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
=! 1111 | 1 1 1 0 | opc1 | 0 | Vd | Rt | 1 0 1 1 | D | opc2 | 1 | (0)(0)(0)(0)

cond

A1

VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>

case opc1:opc2 of
  when '1xxx' advsimd = TRUE; esize = 8; index = UInt(opc1<0>:opc2);
  when '0xx1' advsimd = TRUE; esize = 16; index = UInt(opc1<0>:opc2<1>);
  when '0x00' advsimd = FALSE; esize = 32; index = UInt(opc1<0>);
  when '0x10' UNDEFINED;
  d = UInt(D:Vd); t = UInt(Rt);
  if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
   1 1 1 0 1 1 1 0 | opc1 | 0 | Vd | Rt | 1 0 1 1 | D | opc2 | 1 | (0)(0)(0)(0)

T1

VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>

case opc1:opc2 of
  when '1xxx' advsimd = TRUE; esize = 8; index = UInt(opc1<0>:opc2);
  when '0xx1' advsimd = TRUE; esize = 16; index = UInt(opc1<0>:opc2<1>);
  when '0x00' advsimd = FALSE; esize = 32; index = UInt(opc1<0>);
  when '0x10' UNDEFINED;
  d = UInt(D:Vd); t = UInt(Rt);
  if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> The data size. It must be one of:

8 encoded as opc1<1> = 1. [x] is encoded in opc1<0>, opc2.
Encoded as opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.

Encoded as opc1<1> = 0, opc2 = 0b00. [x] is encoded in opc1<0>.

Equivalent to 32.

The scalar. The register <Dd> is encoded in D:Vd. For details of how [x] is encoded, see the description of <size>.

The source general-purpose register.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    Elem[D[d],index,esize] = R[t]<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Copy a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in a 32-bit SIMD&FP register to a general-purpose register, or the value held in a general-purpose register to a 32-bit SIMD&FP register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  != 1111  1  1  1  0  0  0  0  op  Vn  Rt  1  0  1  0  N[0][0][0]  1[0][0][0][0]
```

From general-purpose register (op == 0)

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

to_arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T1

```
15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
1  1  1  0  1  1  1  0  0  0  0  op  Vn  Rt  1  0  1  0  N[0][0][0]  1[0][0][0][0]
```

From general-purpose register (op == 0)

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

to_arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- <Rt> Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.
- <Sn> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
- <c> See Standard assembler syntax fields.
- <q> See Standard assembler syntax fields.
Operation

if `ConditionPassed()` then
  EncodingSpecificOperations();  `CheckVFPEnabled(TRUE);`
  if to_arm_register then
    $R[t] = S[n];$
  else
    $S[n] = R[t];$
end if

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Copy a vector element to a general-purpose register with sign or zero extension copies a byte, halfword, or word from an Advanced SIMD scalar to a general-purpose register. Bytes and halfwords can be either zero-extended or sign-extended.

On a Floating-point-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see Advanced SIMD scalars. Depending on settings in the CPACR, NSACR, HCPTR, and FPExc registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[ != 1111 \quad 1 \quad 1 \quad 1 \quad 0 \quad U \quad \text{opc1} \quad 1 \quad \text{Vn} \quad \text{Rt} \quad 1 \quad 0 \quad 1 \quad 1 \quad N \quad \text{opc2} \quad 1 \quad (0)(0)(0)(0) \]\n
cond

A1

VMOV{<c>}{<q>}{<dt>} <Rt>, <Dn[x]>

case U:opc1:opc2 of
  when 'x1xxx' advsimd = TRUE; esize = 8; index = UInt(opc1<0>:opc2);
  when 'x0xx1' advsimd = TRUE; esize = 16; index = UInt(opc1<0>:opc2<1>);
  when '00x00' advsimd = FALSE; esize = 32; index = UInt(opc1<0>);
  when '10x00' UNDEFINED;
  when 'x0x10' UNDEFINED;
  t = UInt(Rt); n = UInt(N:Vn); unsigned = (U == '1');
  if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[ 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad U \quad \text{opc1} \quad 1 \quad \text{Vn} \quad \text{Rt} \quad 1 \quad 0 \quad 1 \quad 1 \quad N \quad \text{opc2} \quad 1 \quad (0)(0)(0)(0) \]\n
T1

VMOV{<c>}{<q>}{<dt>} <Rt>, <Dn[x]>

case U:opc1:opc2 of
  when 'x1xxx' advsimd = TRUE; esize = 8; index = UInt(opc1<0>:opc2);
  when 'x0xx1' advsimd = TRUE; esize = 16; index = UInt(opc1<0>:opc2<1>);
  when '00x00' advsimd = FALSE; esize = 32; index = UInt(opc1<0>);
  when '10x00' UNDEFINED;
  when 'x0x10' UNDEFINED;
  t = UInt(Rt); n = UInt(N:Vn); unsigned = (U == '1');
  if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> The data type. It must be one of:
S8
Encoded as U = 0, opc1<1> = 1. [x] is encoded in opc1<0>, opc2.

S16
Encoded as U = 0, opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2.<1>.

U8
Encoded as U = 1, opc1<1> = 1. [x] is encoded in opc1<0>, opc2.

U16
Encoded as U = 1, opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2.<1>.

32
Encoded as U = 0, opc1<1> = 0, opc2 = 0b00. [x] is encoded in opc1<0>.

omitted
Equivalent to 32.

<Rt> The destination general-purpose register.

<Dn[x]> The scalar. For details of how [x] is encoded see the description of <dt>.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if unsigned then
        R[t] = ZeroExtend(Elem[D][n],index,esize], 32);
    else
        R[t] = SignExtend(Elem[D][n],index,esize], 32);
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
VMOV (between two general-purpose registers and two single-precision registers)

Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers transfers the contents of two consecutively numbered single-precision Floating-point registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision Floating-point registers. The general-purpose registers do not have to be contiguous.

Depending on settings in the CPACR, NSACR, HCPR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>cond</th>
<th>Rt2</th>
<th>Rt</th>
<th>M</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>1 1 0 0 0 1 0 op</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From general-purpose registers (op == 0)

VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>

To general-purpose registers (op == 1)

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>

to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If to_arm_registers && t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If m == 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

T1

<table>
<thead>
<tr>
<th>cond</th>
<th>Rt2</th>
<th>Rt</th>
<th>M</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 0 0 0 1 0 op</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
From general-purpose registers (op == 0)

\[ \text{VMOV}\{<c>\}\{<q>\} <Sm>, <Sm1>, <Rt>, <Rt2> \]

To general-purpose registers (op == 1)

\[ \text{VMOV}\{<c>\}\{<q>\} <Rt>, <Rt2>, <Sm>, <Sm1> \]

to_arm_registers = (op == '1');
\[ t = \text{UInt}(Rt); \ t2 = \text{UInt}(Rt2); \ m = \text{UInt}(Vm:M); \]
\[ \text{if } t == 15 \ || \ t2 == 15 \ || \ m == 31 \text{ then UNPREDICTABLE;} \]
\[ \text{if } \text{to_arm_registers } \&\& \ t == t2 \text{ then UNPREDICTABLE;} \]

CONSTRAINED UNPREDICTABLE behavior

If to_arm_registers && t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If m == 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VMOV (between two general-purpose registers and two single-precision registers).

Assembler Symbols

- \(<Rt2>\) Is the second general-purpose register that \(<Sm1>\) will be transferred to or from, encoded in the "Rt2" field.
- \(<Rt>\) Is the first general-purpose register that \(<Sm>\) will be transferred to or from, encoded in the "Rt" field.
- \(<Sm1>\) Is the 32-bit name of the second SIMD&FP register to be transferred. This is the next SIMD&FP register after \(<Sm>\).
- \(<Sm>\) Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Vm:M" field.
- \(<c>\) See Standard assembler syntax fields.
- \(<q>\) See Standard assembler syntax fields.

Operation

if ConditionPassed() then
\[
\begin{align*}
\text{EncodingSpecificOperations(); CheckVFPEnabled(TRUE);} \\
\text{if to_arm_registers then} \\
\quad R[t] &= S[m]; \\
\quad R[t2] &= S[m+1]; \\
\text{else} \\
\quad S[m] &= R[t]; \\
\quad S[m+1] &= R[t2]; 
\end{align*}
\]

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

VMOV (between two general-purpose registers and two single-precision registers)
VMOV (register, SIMD)

Copy between SIMD registers copies the contents of one SIMD register to another.

This is an alias of VORR (register). This means:

- The encodings in this description are named to match the encodings of VORR (register).
- The description of VORR (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}{<dt>} <Dd>, <Dm>

is equivalent to

VORR{<c>}{<q>}{<dt>} <Dd>, <Dm>, <Dm>

and is the preferred disassembly when N:Vn == M:Vm.

128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}{<dt>} <Qd>, <Qm>

is equivalent to

VORR{<c>}{<q>}{<dt>} <Qd>, <Qm>, <Qm>

and is the preferred disassembly when N:Vn == M:Vm.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | D | 1 | 0 | Vn | Vd | 0 | 0 | 0 | 1 | N | Q | M | 1 | Vm |

64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}{<dt>} <Dd>, <Dm>

is equivalent to

VORR{<c>}{<q>}{<dt>} <Dd>, <Dm>, <Dm>

and is the preferred disassembly when N:Vn == M:Vm.

128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}{<dt>} <Qd>, <Qm>

is equivalent to

VORR{<c>}{<q>}{<dt>} <Qd>, <Qm>, <Qm>

and is the preferred disassembly when N:Vn == M:Vm.
Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

&q> See *Standard assembler syntax fields*.

<dt> An optional data type. <dt> must not be F64, but it is otherwise ignored.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

Operation

The description of *VORR (register)* gives the operational pseudocode for this instruction.
Vector Move Long takes each element in a doubleword vector, sign or zero-extends them to twice their original length, and places the results in a quadword vector.

Depending on settings in the \texttt{CPACR}, \texttt{NSACR}, and \texttt{HCPTR} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support}.

It has encodings from the following instruction sets: A32 (\texttt{A1}) and T32 (\texttt{T1}).

\begin{verbatim}
A1

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

if imm3H == '000' then SEE "Related encodings";
if imm3H != '001' && imm3H != '010' && imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);

T1

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

if imm3H == '000' then SEE "Related encodings";
if imm3H != '001' && imm3H != '010' && imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);

Related encodings: See \textit{Advanced SIMD one register and modified immediate} for the T32 instruction set, or \textit{Advanced SIMD one register and modified immediate} for the A32 instruction set.

Assembler Symbols

\begin{verbatim}
<c>
For encoding A1: see \textit{Standard assembler syntax fields}. This encoding must be unconditional.
For encoding T1: see \textit{Standard assembler syntax fields}.

\end{verbatim}

\begin{verbatim}
<q>
See \textit{Standard assembler syntax fields}.

\end{verbatim}

\begin{verbatim}
<dt>
Is the data type for the elements of the operand, encoded in "U:imm3H":

\begin{tabular}{|c|c|}
\hline
U & imm3H \\
\hline
0 & 001 & S8 \\
0 & 010 & S16 \\
0 & 100 & S32 \\
1 & 001 & U8 \\
1 & 010 & U16 \\
1 & 100 & U32 \\
\hline
\end{tabular}

\end{verbatim}

\end{verbatim}

VMOVL
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[Din][m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instructions VRSHRN (zero) and VSHRN (zero).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
1 1 1 1 0 0 1 1 1 | D | 1 1 | size | 1 0 | Vd | 0 0 1 0 0 | M | 0 | Vm
```

A1

\[
\text{VMOVN}\{<c>\}\{<q>\}.<dt> <Dd>, <Qm>
\]

if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm);

T1

```
1 1 1 1 1 1 1 1 | D | 1 1 | size | 1 0 | Vd | 0 0 1 0 0 | M | 0 | Vm
```

T1

\[
\text{VMOVN}\{<c>\}\{<q>\}.<dt> <Dd>, <Qm>
\]

if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm);

Assembler Symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<dt>\) Is the data type for the elements of the operand, encoded in “size”:

\[
\begin{array}{|c|}
\hline
\text{size} & \text{<dt>} \\
\hline
00 & I16 \\
01 & I32 \\
10 & I64 \\
11 & RESERVED \\
\hline
\end{array}
\]

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Qm>\) Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        \[ \text{Elem}[0[d],e,esize] = \text{Elem}[Qin[m>>1],e,2^esize]<esize-1:0>; \]

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Move extraction. This instruction copies the upper 16 bits of the 32-bit source SIMD&FP register into the lower 16 bits of the 32-bit destination SIMD&FP register, while clearing the remaining bits to zero. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**
(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 1 1 1 1 1 0 1 | D 1 1 0 0 0 0 | Vd | 1 0 | 1 0 | 0 | 1 | M | 0 | Vm |

**A1**

VMOVX{<q>}.F16 <Sd>, <Sm>

if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);

**T1**
(Armv8.2)

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 1 1 1 1 1 0 1 | D 1 1 0 0 0 0 | Vd | 1 0 | 1 0 | 0 | 1 | M | 0 | Vm |

**T1**

VMOVX{<q>}.F16 <Sd>, <Sm>

if InITBlock() then UNPREDICTABLE;
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler Symbols**

- **<q>** See *Standard assembler syntax fields*.
- **<Sd>** Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- **<Sm>** Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

**Operation**

if ConditionPassed() then
   EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
   S[d] = zeros(16) : S[m]<31:16>;

VMOVX
VMRS

Move SIMD&FP Special register to general-purpose register moves the value of an Advanced SIMD and floating-point System register to a general-purpose register. When the specified System register is the FPSCR, a form of the instruction transfers the FPSCR.\{N, Z, C, V\} condition flags to the APSR.\{N, Z, C, V\} condition flags.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

When these settings permit the execution of floating-point and Advanced SIMD instructions, if the specified floating-point System register is not the FPSCR, the instruction is UNDEFINED if executed in User mode.

In an implementation that includes EL2, when HCR.TID0 is set to 1, any VMRS access to FPSID from a Non-secure EL1 mode that would be permitted if HCR.TID0 was set to 0 generates a Hyp Trap exception. For more information, see ID group 0, Primary device identification registers.

For simplicity, the VMRS pseudocode does not show the possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{cccccccccccccccc}
\hline
1111 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & \text{reg} & \text{Rt} & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]

cond

A1

VMRS\{<c}\{<q}\}<Rt>, <spec_reg>

\[t = \text{UInt}(\text{Rt});\]
\[\text{if } \!(\text{reg IN }\{'000x', '0101', '01lx', '1000'\}) \text{ then UNPREDICTABLE;};\]
\[\text{if } t == 15 \&\& \text{reg } != '0001' \text{ then UNPREDICTABLE}; // Armv8-A removes UNPREDICTABLE for R13\]

CONSTRANGED UNPREDICTABLE behavior

If \!(\text{reg IN }\{'000x', '0101', '01lx', '1000'\}), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the APSR.\{N, Z, C, V\} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the Rt field, R0 - R14.

T1

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\hline
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & \text{reg} & \text{Rt} & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]

T1

VMRS\{<c}\{<q}\}<Rt>, <spec_reg>

\[t = \text{UInt}(\text{Rt});\]
\[\text{if } \!(\text{reg IN }\{'000x', '0101', '01lx', '1000'\}) \text{ then UNPREDICTABLE;};\]
\[\text{if } t == 15 \&\& \text{reg } != '0001' \text{ then UNPREDICTABLE}; // Armv8-A removes UNPREDICTABLE for R13\]

CONSTRANGED UNPREDICTABLE behavior

If \!(\text{reg IN }\{'000x', '0101', '01lx', '1000'\}), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the APSR.\{N, Z, C, V\} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the Rt field, R0 - R14.

For more information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<
c> See Standard assembler syntax fields.
<q> See Standard assembler syntax fields.
<Rt> Is the general-purpose destination register, encoded in the "Rt" field. Is one of:

- R0-R14
  General-purpose register.

- APSR_nzcv
  Permitted only when <spec_reg> is FPSCR. Encoded as 0b1111. The instruction transfers the FPSCR.\{N, Z, C, V\} condition flags to the APSR.\{N, Z, C, V\} condition flags.

<spec_reg>
Is the source Advanced SIMD and floating-point System register, encoded in “reg”:

<table>
<thead>
<tr>
<th>reg</th>
<th>&lt;spec_reg&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>FPSID</td>
</tr>
<tr>
<td>0001</td>
<td>FPSCR</td>
</tr>
<tr>
<td>001x</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0100</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0101</td>
<td>MVFR2</td>
</tr>
<tr>
<td>0110</td>
<td>MVFR1</td>
</tr>
<tr>
<td>0111</td>
<td>MVFR0</td>
</tr>
<tr>
<td>1000</td>
<td>FPEXC</td>
</tr>
<tr>
<td>1001</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>101x</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>11xx</td>
<td>UNPREDICTABLE</td>
</tr>
</tbody>
</table>

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  if reg == '0001' then  // FPSCR
    CheckVFPEnabled(TRUE);
    if t == 15 then
    else
      R[t] = FPSCR;
  elsif PSTATE.EL == EL0 then
    UNDEFINED;            // Non-FPSCR registers accessible only at PL1 or above
  else
    CheckVFPEnabled(FALSE);  // Non-FPSCR registers are not affected by FPEXC.EN

AArch32.CheckAdvSIMDorFPRegisterTraps(reg);
  case reg of
    when '0000'  R[t] = FPSID;
    when '0101'  R[t] = MVFR2;
    when '0110'  R[t] = MVFR1;
    when '0111'  R[t] = MVFR0;
    when '1000'  R[t] = FPEXC;
    otherwise Unreachable();  // Dealt with above or in encoding-specific pseudocode

VMSR

Move general-purpose register to SIMD&FP Special register moves the value of a general-purpose register to a floating-point System register.
Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode.
For more information see Enabling Advanced SIMD and floating-point support.

When these settings permit the execution of floating-point and Advanced SIMD instructions:
- If the specified floating-point System register is FPSID or FPEXC, the instruction is UNDEFINED if executed in User mode.
- If the specified floating-point System register is the FPSID and the instruction is executed in a mode other than User mode, the instruction is ignored.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|
| != 1111 | 1 1 1 0 1 1 1 0 | reg |
|        | Rt | 1 0 1 0 |(0)|(0)|(0)| 1 |(0)|(0)|(0)|(0) |

cond

VMSR{<c>{<q}> spec_reg}, <Rt>

\[
t = \text{UInt}(\text{Rt}); \]
if reg != '000x' && reg != '1000' then
    Constraint c = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF
            UNDEFINED;
        when Constraint_NOP
            EndOfInstruction();
    if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONstrained UNPREDICTABLE behavior

If reg != '000x' && reg != '1000', then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|
| 1 1 1 0 1 1 1 0 | reg |
| Rt | 1 0 1 0 |(0)|(0)|(0)| 1 |(0)|(0)|(0)|(0) |
T1

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>

t = UInt(Rt);
if reg != '000x' && reg != '1000' then
    Constraint c = ConstrainUnpredictable(Unpredictable_VMSR);
assert c IN {Constraint_UNDEF, Constraint_NOP};
case c of
    when Constraint_UNDEF
        UNDEFINED;
    when Constraint_NOP
        EndOfInstruction();
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If reg != '000x' && reg != '1000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<spec_reg> Is the destination Advanced SIMD and floating-point System register, encoded in "reg":

<table>
<thead>
<tr>
<th>reg</th>
<th>&lt;spec_reg&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>FPSID</td>
</tr>
<tr>
<td>0001</td>
<td>FPSCR</td>
</tr>
<tr>
<td>001x</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>01xx</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>1000</td>
<td>FPEXC</td>
</tr>
<tr>
<td>1001</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>101x</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>11xx</td>
<td>UNPREDICTABLE</td>
</tr>
</tbody>
</table>

<Rt> Is the general-purpose source register, encoded in the "Rt" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
if reg == '0001' then // FPSCR
    CheckVFPEnabled(TRUE);
    FPSCR = R[t];
elsif PSTATE.EL == EL0 then
    UNDEFINED; // Non-FPSCR registers accessible only at PL1 or above
else
    CheckVFPEnabled(FALSE); // Non-FPSCR registers are not affected by FPEXC.EN
    case reg of
        when '0000' // VMSR access to FPSID is ignored
            FPEXC = R[t];
        otherwise Unreachable(); // Dealt with above or in encoding-specific pseudocode
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VMUL (floating-point)

Vector Multiply multiplies corresponding elements in two vectors, and places the results in the destination vector. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

64-bit SIMD vector (Q == 0)

VMUL{<c>}{<q>}={<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMUL{<c>}{<q>}={<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

A2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 1 1 1 0 0 | D | 1 | 0 | Vn | Vd | 1 0 | size | N | 0 | M | 0 | Vm |

cond

Half-precision scalar (size == 01)

(Armv8.2)

VMUL{<c>}{<q>}.F16 {<Sd>}, }<Sn>, <Sm>

Single-precision scalar (size == 10)

VMUL{<c>}{<q>}.F32 {<Sd>}, }<Sn>, <Sm>

Double-precision scalar (size == 11)

VMUL{<c>}{<q>}.F64 {<Dd>}, }<Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D);  n = UInt(Vn:N);  m = UInt(Vm:M);  
  when '10' esize = 32; d = UInt(Vd:D);  n = UInt(Vn:N);  m = UInt(Vm:M);  
  when '11' esize = 64; d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  

VMUL (floating-point)  Page 1008
**CONCONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T1**

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 0 | D | 0 | sz | Vn | Vd | 1 1 0 1 | N | Q | M | 1 | Vm

**64-bit SIMD vector (Q == 0)**

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if sz == '1' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**CONCONSTRAINED UNPREDICTABLE behavior**

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T2**

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 0 1 1 1 0 | D | 1 0 | Vn | Vd | 1 0 | size | N | 0 | M | 0 | Vm

VMUL (floating-point)
Half-precision scalar (size == 01)
(Armv8.2)

```
VMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

if size == '01' && \texttt{InITBlock}() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !\texttt{HaveFP16Ext}()) then UNDEFINED;
advsimd = FALSE;
case size of
  when '01' esize = 16; 
    d = \texttt{UInt}(Vd:D); 
    n = \texttt{UInt}(Vn:N); 
    m = \texttt{UInt}(Vm:M);
  when '10' esize = 32; 
    d = \texttt{UInt}(Vd:D); 
    n = \texttt{UInt}(Vn:N); 
    m = \texttt{UInt}(Vm:M);
  when '11' esize = 64; 
    d = \texttt{UInt}(D:Vd); 
    n = \texttt{UInt}(N:Vn); 
    m = \texttt{UInt}(M:Vm);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && \texttt{InITBlock}(), then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

\texttt{<c>}
\quad For encoding A1: see \textit{Standard assembler syntax fields}. This encoding must be unconditional.
\quad For encoding A2, T1 and T2: see \textit{Standard assembler syntax fields}.

\texttt{<q>}
\quad See \textit{Standard assembler syntax fields}.

\texttt{<dt>}
\quad Is the data type for the elements of the vectors, encoded in “sz”:

\begin{tabular}{|c|c|}
\hline
\texttt{sz} & \texttt{<dt>} \\
\hline
0 & F32 \\
1 & F16 \\
\hline
\end{tabular}

\texttt{<Qd>}
\quad Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \texttt{<Qd>}*2.

\texttt{<Qn>}
\quad Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \texttt{<Qn>}*2.

\texttt{<Qm>}
\quad Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \texttt{<Qm>}*2.

\texttt{<Dd>}
\quad Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\texttt{<Dn>}
\quad Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\texttt{<Dm>}
\quad Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

\texttt{<Sd>}
\quad Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

\texttt{<Sn>}
\quad Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

\texttt{<Sm>}
\quad Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDorVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRVal);
    else  // VFP instruction
        case esize of
            when 16
                S[d] = Zeros(16) : FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            when 32
                S[d] = FPMul(S[n], S[m], FPSCR[]);
            when 64
                D[d] = FPMul(D[n], D[m], FPSCR[]);

VMUL (integer and polynomial)

Vector Multiply multiplies corresponding elements in two vectors. For information about multiplying polynomials see *Polynomial arithmetic over \{0, 1\}*. Depending on settings in the CPACR, NSACR, and HCPTCR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

\[
\text{VMUL}\{<c>\}<q>\} \{<Dd>, }<Dn>, <Dm>
\]

128-bit SIMD vector (Q == 1)

\[
\text{VMUL}\{<c>\}<q>\} \{<Qd>, }<Qn>, <Qm>
\]

T1

```
|   15 |  14 |  13 |  12 |  11 |  10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 1 | 1 | 1 | 0 | D | size | | Vn | Vd | 1 | 0 | 0 | 1 | N | Q | M | 1 | Vm |
```

64-bit SIMD vector (Q == 0)

\[
\text{VMUL}\{<c>\}<q>\} \{<Dd>, }<Dn>, <Dm>
\]

128-bit SIMD vector (Q == 1)

\[
\text{VMUL}\{<c>\}<q>\} \{<Qd>, }<Qn>, <Qm>
\]

For more information about the constrained unpredictable behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- `<c>`
  - For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  - For encoding T1: see *Standard assembler syntax fields*.
- `<q>`
  - See *Standard assembler syntax fields*.
- `<dt>`
  - Is the data type for the elements of the operands, encoded in "op:size":

VMUL (integer and polynomial)
<table>
<thead>
<tr>
<th>op</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>I8</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>I16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>I32</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>P8</td>
</tr>
</tbody>
</table>

**<Qd>** Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

**<Qn>** Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

**<Qm>** Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**<Dn>** Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

**<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize]; op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize]; op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<<(2*esize-1:0);
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Multiply multiplies each element in a vector by a scalar, and places the results in a second vector. For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

**64-bit SIMD vector (Q == 0)**

VMUL{<c>{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]}

**128-bit SIMD vector (Q == 1)**

VMUL{<c>{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]}

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td></td>
</tr>
</tbody>
</table>

**64-bit SIMD vector (Q == 0)**

VMUL{<c>{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]}

**128-bit SIMD vector (Q == 1)**

VMUL{<c>{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]}

**CONSTRAINED UNPREDICTABLE behavior**

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>
For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<dt>
Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

<table>
<thead>
<tr>
<th>F</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>I16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>I32</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>
Is the 64-bit name of the second SIMD&FP source register. When <dt> is I16 or F16, this is encoded in the "Vm<2:0>" field. Otherwise it is encoded in the "Vm" field.

<index>
Is the element index. When <dt> is I16 or F16, this is in the range 0 to 3 and is encoded in the "M:Vm<3>" field. Otherwise it is in the range 0 to 1 and is encoded in the "M" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  op2 = Elem[Din[m], index, esize]; op2val = Int(op2, unsigned);
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[Din[n+r], e, esize]; op1val = Int(op1, unsigned);
      if floating point then
        Elem[D[d+r], e, esize] = FPMul(op1, op2, StandardFPSCRValue());
      else
        if long destination then
          Elem[Q[d+r], e, 2*esize] = (op1val*op2val)<2*esize-1:0>;
        else
          Elem[Q[d+r], e, esize] = (op1val*op2val)<esize-1:0>;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VMULL (integer and polynomial)

Vector Multiply Long multiplies corresponding elements in two vectors. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see Polynomial arithmetic over \{0, 1\}.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 U 1 D != 11 Vn Vd 1 1 op 0 N 0 M 0 Vm
```

**T1**

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 U 1 1 1 1 1 D != 11 Vn Vd 1 1 op 0 N 0 M 0 Vm
```

**CONSTRAINED UNPREDICTABLE behavior**

If op == '1' && size == '10' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the operands, encoded in "op:U:size":

<table>
<thead>
<tr>
<th>op</th>
<th>U</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>U8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>U16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>U32</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>P8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>P64</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];  op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Multiply Long multiplies each element in a vector by a scalar, and places the results in a second vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see Advanced SIMD scalars.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------|-------|-------|-------|-------|
| 1 1 1 1 0 0 1 | U | 1 | D |!=| 11 | Vn | Vd | 1 0 1 0 | N | 1 | M | 0 | Vm |

size

A1

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

if size == '11' then SEE “Related encodings”; 
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------|-------|-------|-------|-------|
| 1 1 1 U | 1 | 1 | 1 | 1 | D |!=| 11 | Vn | Vd | 1 0 1 0 | N | 1 | M | 0 | Vm |

size

T1

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

if size == '11' then SEE “Related encodings”; 
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

See Standard assembler syntax fields.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 01</td>
<td>S16</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
<td></td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
<td></td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
<td></td>
</tr>
</tbody>
</table>
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D.Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16 or U16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16 or U16, otherwise in range 0 to 1, encoded in the "M" field.

**Operation**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VMVN (immediate)

Vector Bitwise NOT (immediate) places the bitwise inverse of an immediate integer constant into every element of the destination register.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1, A2 and A3) and T32 (T1, T2 and T3).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm3</td>
<td>Vd</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

A2

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>i</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm3</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

A3

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>i</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm3</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
64-bit SIMD vector (Q == 0)
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

if (cmode<0> == '1' & cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' & Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

<table>
<thead>
<tr>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>cmode</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

if (cmode<0> == '1' & cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' & Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

<table>
<thead>
<tr>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)
VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)
VMVN{<c>}{<q>}.I16 <Qd>, #<imm>

if (cmode<0> == '1' & cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' & Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

<table>
<thead>
<tr>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
</tbody>
</table>
64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

if (cmode<0> == '1' & cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' & Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEnable{'1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c>    For encoding A1, A2 and A3: see Standard assembler syntax fields. This encoding must be unconditional.
        For encoding T1, T2 and T3: see Standard assembler syntax fields.

<q>    See Standard assembler syntax fields.

<Qd>   Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd>   Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<imm>  Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

Operation

if ConditionPassed() then
   EncodingSpecificOperations();  CheckAdvSIMDEnable();
   for r = 0 to regs-1
      D[d+r] = NOT(imm64);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Bitwise NOT (register) takes a value from a register, inverts the value of each bit, and places the result in the destination register. The registers can be either doubleword or quadword. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

|   | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | D | 1 | 1 | size | 0 | 0 | Vd | 0 | 1 | 0 | 1 | 1 | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>

if size != '00' then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

|   | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | size | 0 | 0 | Vd | 0 | 1 | 0 | 1 | 1 | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>

if size != '00' then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**Assembler Symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  For encoding T1: see *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.
- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(D[m+r]);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit. Depending on settings in the CPACR, NSACR, HCPT, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

31 30 29 28 27 26 25 24 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 | D | 1 | 1 | size | 0 | 1 | Vd | F | 1 | 1 | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VNEG{<c>}{<q>}.dt <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VNEG{<c>}{<q>}.dt <Dd>, <Dm>

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

!= 1111 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 0 | 0 | 0 | 1 | Vd | 1 | 0 | size | 0 | 1 | M | 0 | Vm |

cond

Half-precision scalar (size == 01)
(Armv8.2)

VNEG{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VNEG{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VNEG{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
CONstrained UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 0 1 | D | 1 1 | size | 0 1 | Vd | 0 | F | 1 1 1 | Q | M | 0 | Vm

64-bit SIMD vector (Q == 0)

VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONstrained UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 1 1 0 1 0 1 | D | 1 1 | 0 0 0 1 | Vd | 1 0 | size | 0 1 | M | 0 | Vm

Half-precision scalar (size == 01)

(Armv8.2)

VNEG{<c>}{<q>}.F16 <Dd>, <Dm>

Single-precision scalar (size == 10)

VNEG{<c>}{<q>}.F32 <Dd>, <Dm>

Double-precision scalar (size == 11)

VNEG{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOE. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
- `<q>` For encoding A2, T1 and T2: see *Standard assembler syntax fields*.
- `<dt>` Is the data type for the elements of the vectors, encoded in "F:size":

<table>
<thead>
<tr>
<th>F size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 01</td>
<td>F16</td>
</tr>
<tr>
<td>1 10</td>
<td>F32</td>
</tr>
</tbody>
</table>

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.`
- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.`
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPNeg(Elem[D[m+r],e,esize]);
                else
                    result = -SInt(Elem[D[m+r],e,esize]);
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
                end
            end
        end
    else // VFP instruction
        case esize of
            when 16 S[d] = Zeros(16) : FPNeg(S[m]<15:0>);
            when 32 S[d] = FPNeg(S[m]);
            when 64 D[d] = FPNeg(D[m]);
        end
    end
end
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Negate Multiply Accumulate multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

Arm recommends that software does not use the VNMLA instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPT, and FPXEC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

cond op

Half-precision scalar (size == 01) (Armv8.2)

VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
</tbody>
</table>
Half-precision scalar (size == 01)
(Armv8.2)

VNMLA\{<c}\{<q\}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VNMLA\{<c}\{<q\}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VNMLA\{<c}\{<q\}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

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

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
Operation

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        case vtype of
            when VFPNegMul_VNMLA S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
            when VFPNegMul_VNMLS S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
            when VFPNegMul_VNMUL S[d] = Zeros(16) : FPNeg(product16);
        when 32
            product32 = FPMul(S[n], S[m], FPSCR[]);
        case vtype of
            when VFPNegMul_VNMLA S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
            when VFPNegMul_VNMLS S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
            when VFPNegMul_VNMUL S[d] = FPNeg(product32);
        when 64
            product64 = FPMul(D[n], D[m], FPSCR[]);
        case vtype of
            when VFPNegMul_VNMLA D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
            when VFPNegMul_VNMLS D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
            when VFPNegMul_VNMUL D[d] = FPNeg(product64);
VNMLS

Vector Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. Depending on settings in the CPACR, NSACR, HCPR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | != 1111 | 1 1 1 0 0 | D | 0 1 | Vn | Vd | 1 0 | size | N | 0 | M | 0 | Vm |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | op |

**Half-precision scalar (size == 01)**
(Armv8.2)

VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

**Single-precision scalar (size == 10)**

VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

**Double-precision scalar (size == 11)**

VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
  when '01' esize = 16; d = UINT(Vd:D); n = UINT(Vn:N); m = UINT(Vm:M);
  when '10' esize = 32; d = UINT(Vd:D); n = UINT(Vn:N); m = UINT(Vm:M);
  when '11' esize = 64; d = UINT(D:Vd); n = UINT(N:Vn); m = UINT(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | D | 0 1 | Vn | Vd | 1 0 | size | N | 0 | M | 0 | Vm |
|  op |
Half-precision scalar (size == 01)
(Armv8.2)
VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;

vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
  when '01' esize = 16; d = Uint(Vd:D); n = Uint(Vn:N); m = Uint(Vm:M);
  when '10' esize = 32; d = Uint(Vd:D); n = Uint(Vn:N); m = Uint(Vm:M);
  when '11' esize = 64; d = Uint(D:Vd); n = Uint(N:Vn); m = Uint(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

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

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
Operation

equation VFPNegMul \{ VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL \};

if ConditionPassed() then
  EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
  case esize of
    when 16
      product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
      case vtype of
        when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
        when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
        when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
    when 32
      product32 = FPMul(S[n], S[m], FPSCR[]);
      case vtype of
        when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
        when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
        when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
    when 64
      product64 = FPMul(D[n], D[m], FPSCR[]);
      case vtype of
        when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
        when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
        when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
Vector Negate Multiply multiplies together two floating-point register values, and writes the negation of the result to the destination register.

Depending on settings in the `CPACR`, `NSACR`, `HCPTR`, and `FPEXC` registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```

VNMUL{<c>}{<q>}.F16 {<Sd>},} <Sn>, <Sm>
```

Half-precision scalar (size == 01) (Armv8.2)

```
VNMUL{<c>}{<q>}.F32 {<Sd>},} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VNMUL{<c>}{<q>}.F64 {<Dd>},} <Dn>, <Dm>
```

Double-precision scalar (size == 11)

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '01' && !HaveFP16Ext() then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = VFPNegMul_VNMUL;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1
Half-precision scalar (size == 01)
(Armv8.2)

\[ \text{VNMUL}\{\text{c}\}\{\text{q}\}\}.F16 \{\text{Sd}\}, \{\text{Sn}\}, \{\text{Sm}\} \]

Single-precision scalar (size == 10)

\[ \text{VNMUL}\{\text{c}\}\{\text{q}\}\}.F32 \{\text{Sd}\}, \{\text{Sn}\}, \{\text{Sm}\} \]

Double-precision scalar (size == 11)

\[ \text{VNMUL}\{\text{c}\}\{\text{q}\}\}.F64 \{\text{Dd}\}, \{\text{Dn}\}, \{\text{Dm}\} \]

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '01' && !HaveFP16Ext() then UNDEFINED;
if size == '01' & InITBlock() then UNPREDICTABLE;
vtype = VFPNegMul_VNMUL;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONstrained UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

\(<\text{c}>\) See Standard assembler syntax fields.
\(<\text{q}>\) See Standard assembler syntax fields.
\(<\text{Sd}>\) Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
\(<\text{Sn}>\) Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
\(<\text{Sm}>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
\(<\text{Dd}>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
\(<\text{Dn}>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
\(<\text{Dm}>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
Operation

definition of VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
                when VFPNegMul_VNMLS S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                when VFPNegMul_VNMUL S[d] = Zeros(16) : FPNeg(product16);
        when 32
            product32 = FPMul(S[n], S[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                when VFPNegMul_VNMLS S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
                when VFPNegMul_VNMUL S[d] = FPNeg(product32);
        when 64
            product64 = FPMul(D[n], D[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                when VFPNegMul_VNMLS D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
                when VFPNegMul_VNMUL D[d] = FPNeg(product64);
VORN (register)

Vector bitwise OR NOT (register) performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>Vn</td>
<td>Vd</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}{<dt>} {<Qd>}, <Qn>, <Dm>

128-bit SIMD vector (Q == 1)

VORN{<c>}{<q>}{<dt>} {<Qd>}, <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

### T1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>Vn</td>
<td>Vd</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}{<dt>} {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VORN{<c>}{<q>}{<dt>} {<Qd>}, <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

### Assembler Symbols

- <c>: For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

  For encoding T1: see [Standard assembler syntax fields](#).

- <q>: See [Standard assembler syntax fields](#).

- <dt>: An optional data type. It is ignored by assemblers, and does not affect the encoding.

- <Qd>: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

- <Qn>: Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

- <Qm>: Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

- <Dd>: Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- <Dn>: Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

- <Dm>: Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR NOT(D[m+r]);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Bitwise OR NOT (immediate) performs a bitwise OR between a register value and the complement of an immediate value, and returns the result into the destination vector.

This is a pseudo-instruction of VORR (immediate). This means:

- The encodings in this description are named to match the encodings of VORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (immediate) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm3</td>
<td>Vd</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>

is equivalent to
VORR{<c>}{<q>}.I16 <Dd>, #~<imm>

128-bit SIMD vector (Q == 1)

VORN{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>

is equivalent to
VORR{<c>}{<q>}.I16 <Qd>, #~<imm>

A2

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm3</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>

is equivalent to
VORR{<c>}{<q>}.I32 <Dd>, #~<imm>

128-bit SIMD vector (Q == 1)

VORN{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>

is equivalent to
VORR{<c>}{<q>}.I32 <Qd>, #~<imm>

T1
64-bit SIMD vector (Q == 0)

\[ VORN\{<c>\}{<q>}\}.I16 \{<Dd>,} <Dd>, #<imm> \]

is equivalent to

\[ VORR\{<c>\}{<q>}\}.I16 <Dd>, #~<imm> \]

128-bit SIMD vector (Q == 1)

\[ VORN\{<c>\}{<q>}\}.I16 \{<Qd>,} <Qd>, #<imm> \]

is equivalent to

\[ VORR\{<c>\}{<q>}\}.I16 <Qd>, #~<imm> \]

T2

64-bit SIMD vector (Q == 0)

\[ VORN\{<c>\}{<q>}\}.I32 \{<Dd>,} <Dd>, #<imm> \]

is equivalent to

\[ VORR\{<c>\}{<q>}\}.I32 <Dd>, #~<imm> \]

128-bit SIMD vector (Q == 1)

\[ VORN\{<c>\}{<q>\}.I32 \{<Qd>,} <Qd>, #<imm> \]

is equivalent to

\[ VORR\{<c>\}{<q>\}.I32 <Qd>, #~<imm> \]

Assembler Symbols

- `<c>`: For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1 and T2: see Standard assembler syntax fields.
- `<q>`: See Standard assembler syntax fields.
- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>^2`.
- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<imm>`: Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

Operation

The description of VORR (immediate) gives the operational pseudocode for this instruction.
VORR (immediate)

Vector Bitwise OR (immediate) performs a bitwise OR between a register value and an immediate value, and returns the result into the destination vector.
Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction **VORN (immediate)**.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

**A1**

64-bit SIMD vector (Q == 0)

\[
\text{VORR}\{<c>\}<q>.I32 \{<Dd>,\} <Dd>, \#<imm>
\]

128-bit SIMD vector (Q == 1)

\[
\text{VORR}\{<c>\}<q>.I32 \{<Qd>,\} <Qd>, \#<imm>
\]

if `cmode<0> == '0' || cmode<3:2> == '11'` then SEE "VMOV (immediate)";
if `Q == '1' && Vd<0> == '1'` then UNDEFINED;
`imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4)`;
`d = UInt(D:Vd);`  `regs = if Q == '0' then 1 else 2;`

**A2**

64-bit SIMD vector (Q == 0)

\[
\text{VORR}\{<c>\}<q>.I16 \{<Dd>,\} <Dd>, \#<imm>
\]

128-bit SIMD vector (Q == 1)

\[
\text{VORR}\{<c>\}<q>.I16 \{<Qd>,\} <Qd>, \#<imm>
\]

if `cmode<0> == '0' || cmode<3:2> == '11'` then SEE "VMOV (immediate)";
if `Q == '1' && Vd<0> == '1'` then UNDEFINED;
`imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4)`;
`d = UInt(D:Vd);`  `regs = if Q == '0' then 1 else 2;`

**T1**

64-bit SIMD vector (Q == 0)

\[
\text{VORR}\{<c>\}<q>.I32 \{<Dd>,\} <Dd>, \#<imm>
\]

128-bit SIMD vector (Q == 1)

\[
\text{VORR}\{<c>\}<q>.I32 \{<Qd>,\} <Qd>, \#<imm>
\]

if `cmode<0> == '0' || cmode<3:2> == '11'` then SEE "VMOV (immediate)";
if `Q == '1' && Vd<0> == '1'` then UNDEFINED;
`imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4)`;
`d = UInt(D:Vd);`  `regs = if Q == '0' then 1 else 2;`
64-bit SIMD vector (Q == 0)

\[ \text{VORR} \{<c>\} \{<q>\}.I32 \{<Dd>,\} \ <Dd>, \ #<imm> \]

128-bit SIMD vector (Q == 1)

\[ \text{VORR} \{<c>\} \{<q>\}.I32 \{<Qd>,\} \ <Qd>, \ #<imm> \]

\[ \begin{align*}
\text{if } \text{cmode}<0> &= '0' \ | \ | \text{cmode}<3:2> == '11' \text{ then } \text{SEE "VMOV (immediate)"}; \\
\text{if } Q &= '1' \ &\& \ Vd<0> == '1' \text{ then UNDEFINED; } \\
\text{imm64} &= \text{AdvSIMDExpandImm}('0', \text{cmode}, i:\text{imm3:imm4}); \\
d &= \text{UInt}(D:Vd); \quad \text{regs} = \text{if } Q == '0' \text{ then 1 else 2;}
\end{align*} \]

T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 0 0 0 imm3</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>cmode</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

\[ \text{VORR} \{<c>\} \{<q>\}.I16 \{<Dd>,\} \ <Dd>, \ #<imm> \]

128-bit SIMD vector (Q == 1)

\[ \text{VORR} \{<c>\} \{<q>\}.I16 \{<Qd>,\} \ <Qd>, \ #<imm> \]

\[ \begin{align*}
\text{if } \text{cmode}<0> &= '0' \ | \ | \text{cmode}<3:2> == '11' \text{ then } \text{SEE "VMOV (immediate)"}; \\
\text{if } Q &= '1' \ &\& \ Vd<0> == '1' \text{ then UNDEFINED; } \\
\text{imm64} &= \text{AdvSIMDExpandImm}('0', \text{cmode}, i:\text{imm3:imm4}); \\
d &= \text{UInt}(D:Vd); \quad \text{regs} = \text{if } Q == '0' \text{ then 1 else 2;}
\end{align*} \]

Assembler Symbols

\(<c>\)For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1 and T2: see Standard assembler syntax fields.

\(<q>\)See Standard assembler syntax fields.

\(<Qd>\)Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>\)*2.

\(<Dd>\)Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<\text{imm}>\)Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of \(<\text{imm}>\), see Modified immediate constants in T32 and A32 Advanced SIMD instructions.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

\[ \text{if } \text{ConditionPassed}() \text{ then } \] \[ \text{EncodingSpecificOperations(); } \text{CheckAdvSIMDEnabled(); } \]
\[ \text{for } r = 0 \text{ to } \text{regs}-1 \] \[ D[d+r] = D[d+r] \text{ OR imm64; } \]

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector bitwise OR (register) performs a bitwise OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the alias VMOV (register, SIMD).

This instruction is used by the pseudo-instructions VRSHR (zero) and VSHR (zero).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

64-bit SIMD vector (Q == 0)

VORR\{\<c)\}\{\<q)\}\{\<dt)\} \{\<Dd),\} \<Dn), \<Dm

128-bit SIMD vector (Q == 1)

VORR\{\<c)\}\{\<q)\}\{\<dt)\} \{\<Qd),\} \<Qn), \<Qm

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

\[d = \text{UInt}(D:Vd); \ n = \text{UInt}(N:Vn); \ m = \text{UInt}(M:Vm); \ \text{regs} = \text{if Q == '0' then 1 else 2};\]

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>Vn</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>N</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VORR\{\<c)\}\{\<q)\}\{\<dt)\} \{\<Dd),\} \<Dn), \<Dm

128-bit SIMD vector (Q == 1)

VORR\{\<c)\}\{\<q)\}\{\<dt)\} \{\<Qd),\} \<Qn), \<Qm

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

\[d = \text{UInt}(D:Vd); \ n = \text{UInt}(N:Vn); \ m = \text{UInt}(M:Vm); \ \text{regs} = \text{if Q == '0' then 1 else 2};\]

Assembler Symbols

\(<c)\) For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

\(<q)\) See Standard assembler syntax fields.

\(<dt)\) An optional data type. It is ignored by assemblers, and does not affect the encoding.

\(<Qd)\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

\(<Qn)\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

\(<Qm)\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

\(<Dd)\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV (register, SIMD)</td>
<td>N:Vn == M:Vm</td>
</tr>
<tr>
<td>VRSHR (zero)</td>
<td>Never</td>
</tr>
<tr>
<td>VSHR (zero)</td>
<td>Never</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR D[m+r];
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector. The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements. The following figure shows an example of the operation of VPADAL doubleword operation for data type S16.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 1 1 1 1 1 1 | D | 1 | 1 | size | 0 | 0 | Vd | 0 | 1 | 1 | 0 | op | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
Assembler Symbols

For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

See *Standard assembler syntax fields*.

Is the data type for the elements of the vectors, encoded in "op:size":

<table>
<thead>
<tr>
<th>op</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>S8</td>
</tr>
<tr>
<td>0 1</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>0 11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
<tr>
<td>1 11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];
            op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = Elem[D[d+r],e,2*esize] + result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Pairwise Add (floating-point) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.
The operand and result elements are floating-point numbers.
Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

VPADD{<c>}{<q>}{<dt>} {<Od>, }<On>, <Dm>

if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |---------------------------------------------------------------|
|--------------------------------------|
| 1 1 1 1 1 1 1 1 0 D 0 sz Vn Vd 1 1 0 1 N Q M 0 Vm            |

T1

VPADD{<c>}{<q>}{<dt>} {<Od>, }<On>, <Dm>

if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>  For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.  
For encoding T1: see Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<dt>  Is the data type for the elements of the vectors, encoded in "sz"
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

- **<Dd>** is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- **<Dn>** is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- **<Dm>** is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        Elemdest,e,esize] = FPAdd(ElemD[n],2*e,esize], Elemdn,2*e+1,esize], StandardFPSCRValue());
        Elemdest,e+h,esize] = FPAdd(ElemD[m],2*e,esize], Elem[m],2*e+1,esize], StandardFPSCRValue());
D[d] = dest;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector. The operands and result are doubleword vectors. The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers. The following figure shows an example of the operation of VPADD doubleword operation for data type I16.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>1 1 1 1 0</th>
<th>D size</th>
<th>Vd</th>
<th>1 0 1 1</th>
<th>N Q M 1</th>
<th>Vm</th>
</tr>
</thead>
</table>

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**Assembler Symbols**

- `<c>` For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1: see Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<dt>` Is the data type for the elements of the vectors, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>I8</td>
</tr>
<tr>
<td>01</td>
<td>I16</td>
</tr>
<tr>
<td>10</td>
<td>I32</td>
</tr>
</tbody>
</table>
<Dd> is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        Elem[dest,e,esize] = Elem[D[n],2*e,esize] + Elem[D[n],2*e+1,esize];
        Elem[dest,e+h,esize] = Elem[D[m],2*e,esize] + Elem[D[m],2*e+1,esize];
    D[d] = dest;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows an example of the operation of VPADDL doubleword operation for data type S16.

Dm<1:<0> + Dd<1:<0>

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see **Enabling Advanced SIMD and floating-point support**.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

64-bit SIMD vector (Q == 0)

VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--- | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | size | 0 | 0 | Vd | 0 | 0 | 1 | 0 | op | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

&q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in "op:size":

<table>
<thead>
<tr>
<th>op</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>U8</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>U16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>U32</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  h = elements DIV 2;
  for r = 0 to regs-1
    for e = 0 to h-1
      op1 = Elem[D][m+r],2*e,esize]; op2 = Elem[D][m+r],2*e+1,esize];
      result = Int(op1, unsigned) + Int(op2, unsigned);
      Elem[D][d+r],e,2*esize] = result<2*esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
VPMAX (floating-point)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 1 1 1 1 0 0 1 1 0 | D 0 | sz | Vn | Vd | 1 1 1 1 | N 0 | M 0 | Vm
   op
```

A1

```
VPMAX{<c>}{<q>}{<dt>}{<Dd>, }<Dn>, <Dm>
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
T1

   15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 1 1 1 1 1 1 1 1 1 0 | D 0 | sz | Vn | Vd | 1 1 1 1 | N 0 | M 0 | Vm
   op
```

T1

```
VPMAX{<c>}{<q>}{<dt>}{<Dd>, }<Dn>, <Dm>
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

CONSTRANDED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- `<c>` For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
- For encoding T1: see Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<dt>` Is the data type for the elements of the vectors, encoded in “sz”:

```
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

< Dt> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

< Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

< Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```markdown
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = if maximum then FPMAX(op1,op2,StandardFPSCRValue()) else FPMIN(op1,op2,StandardFPSCRValue());
        op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = if maximum then FPMAX(op1,op2,StandardFPSCRValue()) else FPMIN(op1,op2,StandardFPSCRValue());
    D[d] = dest;
```

Internal version only: isa v01_19, pseudocode v2020-09 xml, sve v2020-09 rc3; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VPMAX (integer)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

The following figure shows an example of the operation of VPMAX doubleword operation for data type S16 or U16.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

\[
\begin{array}{c}
\text{VPMAX}\{<c>\}\{<q>\}\{<dt>\} \{<Dd>, \}<Dn>, \langle Dm> \\
\text{if size == '11' then UNDEFINED;}
\text{maximum = (op == '0'); unsigned = (U == '1');}
\text{esize = 8 \ll \text{UInt}(size); elements = 64 \div \text{esize};}
\text{d = \text{UInt}(D:Vd); n = \text{UInt}(N:Vn); m = \text{UInt}(M:Vm);}
\end{array}
\]

T1

\[
\begin{array}{c}
\text{VPMAX}\{<c>\}\{<q>\}\{<dt>\} \{<Dd>, \}<Dn>, \langle Dm> \\
\text{if size == '11' then UNDEFINED;}
\text{maximum = (op == '0'); unsigned = (U == '1');}
\text{esize = 8 \ll \text{UInt}(size); elements = 64 \div \text{esize};}
\text{d = \text{UInt}(D:Vd); n = \text{UInt}(N:Vn); m = \text{UInt}(M:Vm);}
\end{array}
\]

Assembler Symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
\(<q>\) See Standard assembler syntax fields.
\(<dt>\) Is the data type for the elements of the operands, encoded in “U:size”: 
### Operation

```c
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;
    D[d] = dest;
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

A1

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | D  | 1  | sz | Vn | Vd | 1  | 1  | 1  | 1  | N  | 0  | M  | 0  | Vm |
| op |

T1

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);

CONSTRANGED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd“ field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<DM> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm“ field.

### Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize]; op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,StandardFPSCRValue());
        op1 = Elem[D[m],2*e,esize]; op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,StandardFPSCRValue());
    D[d] = dest;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VPMIN (integer)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
VPMIN{<c>}{<q>}{<dt>}{<Dd>, }<Dn>, <Dm>
```

if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

### T1

```
VPMIN{<c>}{<q>}{<dt>}{<Dd>, }<Dn>, <Dm>
```

if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m =UInt(M:Vm);

### Assembler Symbols

- `<c>` For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
- `<q>` For encoding T1: see Standard assembler syntax fields.
- `<dt>` Is the data type for the elements of the operands, encoded in "U:size":

<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
</tbody>
</table>

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
Operation

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDEnabled();
   bits(64) dest;
   h = elements DIV 2;

   for e = 0 to h-1
      op1 = Int(Elem[D[n],2*e,esize], unsigned);
      op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
      result = if maximum then Max(op1,op2) else Min(op1,op2);
      Elem[dest,e,esize] = result<esize-1:0>;
      op1 = Int(Elem[D[m],2*e,esize], unsigned);
      op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
      result = if maximum then Max(op1,op2) else Min(op1,op2);
      Elem[dest,e+h,esize] = result<esize-1:0>;

   D[d] = dest;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Pop SIMD&FP registers from Stack loads multiple consecutive Advanced SIMD and floating-point register file registers from the stack.

This is an alias of `VLDM, VLDMDB, VLDMIA`. This means:

- The encodings in this description are named to match the encodings of `VLDM, VLDMDB, VLDMIA`.
- The description of `VLDM, VLDMDB, VLDMIA` gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

```
cond   P  U  W   Rn
imm8<7:1> 1 0 1 1
Vd

! = 1111 1 1 0 0 1 1 1 1 0 1
imm8<0>
```

Increment After

```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

A2

```
cond   P  U  W   Rn
imm8   1 0 1 1
Vd

! = 1111 1 1 0 0 1 1 1 1 0 1
imm8<0>
```

Increment After

```
VPOP{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

T1

```
cond   P  U  W   Rn
imm8<7:1> 1 0 1 1
Vd

1 1 1 0 1 1 0 0 1 1 1 1 0 1
imm8<0>
```

Increment After

```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.
Increment After

VPOP{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<sreglist> Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

<dreglist> Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Operation

The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.
VPUSH

Push SIMD&FP registers to Stack stores multiple consecutive registers from the Advanced SIMD and floating-point register file to the stack.

This is an alias of VSTM, VSTMDB, VSTMIA. This means:

- The encodings in this description are named to match the encodings of VSTM, VSTMDB, VSTMIA.
- The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

| 31 30 29 28 27 26 25 24 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 1 0 1 1 0 1     | Vd 1 0 1 1 | imm8<7:1>      |
| cond P U W Rn                   | imm8<0>      |

Decrement Before

VPUSH{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

A2

| 31 30 29 28 27 26 25 24 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 1 0 1 1 0 1     | Vd 1 0 1 0 | imm8           |
| cond P U W Rn                   | imm8<0>      |

Decrement Before

VPUSH{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------------------|-----------------|
| 1 1 1 0 1 1 0 1 0 1 0 1 1 0 1     | Vd 1 0 1 1 | imm8<7:1>      |
| P U W Rn                          | imm8<0>      |

Decrement Before

VPUSH{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.
Decrement Before

VPUSH{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VSTMDB{<c>}{<q}>{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<sreglist> Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

<dreglist> Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Operation

The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.
VQABS

Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, `FPSCR.QC`, is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the `CPACR`, `NSACR`, and `HCPTR` registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

64-bit SIMD vector \((Q == 0)\)

\[
\text{VQABS}\{<c>\}\{<q>\}.<dt> <Dd>, <Dm>
\]

128-bit SIMD vector \((Q == 1)\)

\[
\text{VQABS}\{<c>\}\{<q>\}.<dt> <Qd>, <Qm>
\]

\[
\text{if } \text{size} == '11' \text{ then UNDEFINED;}
\]

\[
\text{if } Q == '1' \&\& (Vd<0> == '1' || Vm<0> == '1') \text{ then UNDEFINED;}
\]

\[
\text{esize} = 8 << \text{UInt}(\text{size}); \text{ elements} = 64 \text{ DIV esize};
\]

\[
d = \text{UInt}(D:\text{Vd}); \ m = \text{UInt}(M:\text{Vm}); \ \text{regs} = \text{if } Q == '0' \text{ then 1 else 2;}
\]

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | Q  | M  | 0  | Vm |

64-bit SIMD vector \((Q == 0)\)

\[
\text{VQABS}\{<c>\}\{<q>\}.<dt> <Dd>, <Dm>
\]

128-bit SIMD vector \((Q == 1)\)

\[
\text{VQABS}\{<c>\}\{<q>\}.<dt> <Qd>, <Qm>
\]

\[
\text{if } \text{size} == '11' \text{ then UNDEFINED;}
\]

\[
\text{if } Q == '1' \&\& (Vd<0> == '1' || Vm<0> == '1') \text{ then UNDEFINED;}
\]

\[
\text{esize} = 8 << \text{UInt}(\text{size}); \text{ elements} = 64 \text{ DIV esize};
\]

\[
d = \text{UInt}(D:\text{Vd}); \ m = \text{UInt}(M:\text{Vm}); \ \text{regs} = \text{if } Q == '0' \text{ then 1 else 2;}
\]

**Assembler Symbols**

\(<c>\) For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: see [Standard assembler syntax fields](#).

\(<q>\) See [Standard assembler syntax fields](#).

\(<dt>\) Is the data type for the elements of the vectors, encoded in “size”:
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Abs(SInt(Elem[D[m+r],e,esize]));
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector. If any of the results overflow, they are saturated. The cumulative saturation bit, \texttt{FPSCR.QC}, is set if saturation occurs. For details see \textit{Pseudocode details of saturation}. Depending on settings in the \texttt{CPACR}, \texttt{NSACR}, and \texttt{HCPTR} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support}.

It has encodings from the following instruction sets: A32 (\texttt{A1}) and T32 (\texttt{T1}).

\begin{verbatim}
A1

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 | U | 0 | D | size | Vn | Vd | 0 0 0 0 | N | Q | M | 1 | Vm
\end{verbatim}

64-bit SIMD vector (Q == 0)

\texttt{VQADD\{<c>\}{<q>}.<dt> \{<Dd>,\} \{<Dn>, \{<Dm>}

128-bit SIMD vector (Q == 1)

\texttt{VQADD\{<c>\}{<q>}.<dt> \{<Qd>,\} <Qn}, <Qm>

\begin{verbatim}
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
\end{verbatim}

\begin{verbatim}
T1

\begin{verbatim}
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 | U | 1 1 1 1 0 | D | size | Vn | Vd | 0 0 0 0 | N | Q | M | 1 | Vm
\end{verbatim}

64-bit SIMD vector (Q == 0)

\texttt{VQADD\{<c>\}{<q>}.<dt> \{<Dd>,\} \{<Dn>, \{<Dm>}

128-bit SIMD vector (Q == 1)

\texttt{VQADD\{<c>\}{<q>}.<dt> \{<Qd>,\} <Qn}, <Qm>

\begin{verbatim}
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
\end{verbatim}

\textbf{Assembler Symbols}

\texttt{<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1: see Standard assembler syntax fields.\texttt{<q> See Standard assembler syntax fields.\texttt{<dt> Is the data type for the elements of the vectors, encoded in “U:size”:}

VQADD
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            sum = Int(Elem[D][n+r],e,esize], unsigned) + Int(Elem[D][m+r],e,esize], unsigned);
            (Elem[D][d+r],e,esize], sat) = SatQ(sum, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VQDMLAL

Vector Saturating Doubling Multiply Accumulate Long multiplies corresponding elements in two doubleword vectors, doubles the products, and accumulates the results into the elements of a quadword vector. The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

if size == '11' then SEE “Related encodings”;
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 8 << UInt(size); elements = 64 DIV esize;

A2

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

if size == '11' then SEE “Related encodings”;
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn);
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);
T1

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 8 << UInt(size); elements = 64 DIV esize;

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn);
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<q> For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1 and T2: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operands, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> For encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
**VQDMLSL**

Vector Saturating Doubling Multiply Subtract Long multiplies corresponding elements in two doubleword vectors, subtracts double the products from corresponding elements of a quadword vector, and places the results in the same quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

**A1**

![Instruction Encoding](image1)

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;

**A2**

![Instruction Encoding](image2)

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

**T1**

![Instruction Encoding](image3)
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields.

Is the data type for the elements of the operands, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.

Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.
Operation

if `ConditionPassed()` then

    `EncodingSpecificOperations()`;
    `CheckAdvSIMDEnabled()`;
    if scalar form then `op2 = SInt(Elem[Din][m],index,esize])`;
    for `e = 0` to `elements-1`
        if !scalar form then `op2 = SInt(Elem[Din][m],e,esize])`;
        `op1 = SInt(Elem[Din][n],e,esize])`;
        // The following only saturates if both `op1` and `op2` equal -(2^(esize-1))
        `product, sat1) = SignedSatQ(2*op1*op2, 2*esize);`
        if add then
            `result = SInt(Elem[Qin][d>>1],e,2*esize]) + SInt(product);`
        else
            `result = SInt(Elem[Qin][d>>1],e,2*esize]) - SInt(product);`
        `((Elem[Q][d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);`
        if sat1 || sat2 then `FPSCR.QC = '1';`
VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated, for rounded results see VQRDMULH.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

|   31  |  30  |  29  |  28  |  27  |  26  |  25  |  24  |  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 | 0 | 0 | D | size | Vn | Vd | 1 0 | 1 1 | N | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VQDMULH{<c>}{<q>}.<dt> {<Qd>,, <Qn>, <Qm>}

128-bit SIMD vector (Q == 1)

VQDMULH{<c>}{<q>}.<dt> {<Qd>,, <Qn>, <Qm>}

if Q == '1' && (Vd<0> == '1' | | Vn<0> == '1' | | Vm<0> == '1') then UNDEFINED;
if size == '00' | | size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

A2

|   31  |  30  |  29  |  28  |  27  |  26  |  25  |  24  |  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 | Q | 1 | D | != 11 | Vn | Vd | 1 0 | 1 0 | N | 1 | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VQDMULH{<c>}{<q>}.<dt> {<Qd>,, <Qn>, <Qm>}

128-bit SIMD vector (Q == 1)

VQDMULH{<c>}{<q>}.<dt> {<Qd>,, <Qn>, <Qm>}

if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' | | Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  index = UInt(M:Vm<3>);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);
if size == '00' then esize = 32;  elements = 2;  m = UInt(Vm);

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 1</td>
<td>0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>1 0</td>
<td>1 1</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
64-bit SIMD vector (Q == 0)

VQDMULH\{<c>\}{<q>\}.<dt> \{<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' then UNDEFINED;
scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
size = '01' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
size = '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
size = '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

\(<c>\) For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields.

\(<q>\) See Standard assembler syntax fields.

\(<dt>\) Is the data type for the elements of the operands, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
</tbody>
</table>

\(<Qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

\(<Qn>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

\(<Qm>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Dm[x]>\) Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
if scalar_form then op2 = SInt(Elem[D][m],index,esize));
for r = 0 to regs-1
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[D][m+r],e,esize));
        op1 = SInt(Elem[D][n+r],e,esize));
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (result, sat) = SignedSatQ((2*op1*op2) >> esize, esize);
        Elem[D][d+r],e,esize] = result;
        if sat then FPSCR.QC = '1';
VQDMULL

Vector Saturating Doubling Multiply Long multiplies corresponding elements in two doubleword vectors, doubles the products, and places the results in a quadword vector. The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
1  1  1  1  0  0  1  0  1  D != 11  Vn  Vd  1  1  0  1  N  0  M  0  Vm
```

size

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;

A2

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
1  1  1  1  0  0  1  0  1  D != 11  Vn  Vd  1  0  1  1  N  1  M  0  Vm
```

size

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

T1

```
15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
1  1  1  0  1  1  1  1  D != 11  Vn  Vd  1  1  0  1  N  0  M  0  Vm
```

size

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
T2

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & D & ! = & 1 & 1 & Vn & Vd & 1 & 0 & 1 & 1 & N & 1 & M & 0 & Vm
\end{array}
\]

size

T2

VQDMULL\(<\text{c}>\}{<\text{q}>}.<\text{dt}> \ <\text{Qd}>$, \ <\text{Dn}>$, \ <\text{Dm}[x]>$

if size == '11' then SEE "Related encodings";
if size == '00' \|| Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = \text{UInt}(D:Vd);  n = \text{UInt}(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = \text{UInt}(Vm<2:0>);  index = \text{UInt}(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = \text{UInt}(Vm);  index = \text{UInt}(M);

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

\(<\text{c}>\)

For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields.

\(<\text{q}>\)

See Standard assembler syntax fields.

\(<\text{dt}>\)

Is the data type for the elements of the operands, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
</tbody>
</table>

\(<\text{Qd}>\)

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<\text{Qd}>*2$.

\(<\text{Dn}>\)

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<\text{Dm}[x]>\)

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If \(<\text{dt}>\) is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If \(<\text{dt}>\) is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

\(<\text{Dm}>\)

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = \text{SInt}(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = \text{SInt}(Elem[Din[m],e,esize]);
        opl = \text{SInt}(Elem[Din[n],e,esize]);
        // The following only saturates if both opl and op2 equal -(2^(esize-1))
        (product, sat) = \text{SignedSatQ}(2*opl*op2, 2*esize);
        Elem[Q[d>>1],e,2*esize] = product;
        if sat then FPSCR.QC = '1';

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**VQMOVN, VQMOVUN**

Vector Saturating Move and Narrow copies each element of the operand vector to the corresponding element of the destination vector.

The operand is a quadword vector. The elements can be any one of:
- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result is a doubleword vector. The elements are half the length of the operand vector elements. If the operand is unsigned, the results are unsigned. If the operand is signed, the results can be signed or unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the pseudo-instructions VQRSHRN (zero), VQRSHRUN (zero), VQSHRN (zero), and VQSHRUN (zero).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

Signed result (op == 1x)

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Unsigned result (op == 01)

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

if op == '00' then SEE "VMOVN";
if size == '11' || Vm<0> == '1' then UNDEFINED;
src unsigned = (op == '11'); dest unsigned = (op<0> == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm);

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 D 1 1 size 1 0 Vd 0 0 1 0 op M 0 Vm</td>
</tr>
</tbody>
</table>

Signed result (op == 1x)

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Unsigned result (op == 01)

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

if op == '00' then SEE "VMOVN";
if size == '11' || Vm<0> == '1' then UNDEFINED;
src unsigned = (op == '11'); dest unsigned = (op<0> == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm);
Assembler Symbols

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

See Standard assembler syntax fields.

For the signed result variant: is the data type for the elements of the operand, encoded in “op<0>:size”:

<table>
<thead>
<tr>
<th>op&lt;0&gt;</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>S32</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S64</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>U16</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>U32</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>U64</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the unsigned result variant: is the data type for the elements of the operand, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>S16</td>
</tr>
<tr>
<td>01</td>
<td>S32</td>
</tr>
<tr>
<td>10</td>
<td>S64</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        (Elem[D[d],e,esize], sat) = SatQ(operand, esize, dest_unsigned);
        if sat then FPSCR.QC = '1';
**VQNEG**

Vector Saturating Negate negates each element in a vector, and places the results in the destination vector. If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

**64-bit SIMD vector (Q == 0)**

VQNEG{<c>{<q>}<dt>} <Dd>, <Dm>

**128-bit SIMD vector (Q == 1)**

VQNEG{<c}{<q}> <dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 * UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | size | 0 | 0 | Vd | 0 | 1 | 1 | 1 | 1 | Q | M | 0 | Vm |

**64-bit SIMD vector (Q == 0)**

VQNEG{<c>{<q>}<dt>} <Dd>, <Dm>

**128-bit SIMD vector (Q == 1)**

VQNEG{<c>{<q}> <dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 * UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

### Assembler Symbols

- `<c>` For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
  
  For encoding T1: see Standard assembler syntax fields.

- `<q>` See Standard assembler syntax fields.

- `<dt>` Is the data type for the elements of the vectors, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>S8</td>
</tr>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = \texttt{SInt}(\texttt{Elem}[D[m+r],e,esize]);
            (\texttt{Elem}[D[d+r],e,esize], sat) = \texttt{SignedSatQ}(result, esize);
            if sat then FPSCR.QC = '1';
VQRDMLAH

Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1
(Armv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------------------------|-----------------|-----------------|-----------------|
| 1 1 1 1 0 0 1 1 0 D size Vn Vd 1 0 1 1 N Q M 1 | Vm | size 64-bit SIMD vector (Q == 0) |
| VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm> |

128-bit SIMD vector (Q == 1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------------------------|-----------------|-----------------|-----------------|
| 1 1 1 1 1 0 0 0 1 | Q | D != 11 | Vn Vd 1 0 1 1 N Q M 1 | Vm |
| size |
| 64-bit SIMD vector (Q == 0) |
| VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]> |

128-bit SIMD vector (Q == 1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------------------------|-----------------|-----------------|-----------------|
| 1 1 1 1 0 0 0 1 | Q | D != 11 | Vn Vd 1 0 1 1 N Q M 0 | Vm |
| size |
| 64-bit SIMD vector (Q == 0) |
| VQRDMLAH{<q>}.<dt> <Dd>, <Qn>, <Qm> |

T1
(Armv8.1)

if !HaveQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);
64-bit SIMD vector \((Q == 0)\)

\[
\text{VQRDMLAH}\{<q>\}.<dt> <Dd>, <Dn>, <Dm>
\]

128-bit SIMD vector \((Q == 1)\)

\[
\text{VQRDMLAH}\{<q>\}.<dt> <Qd>, <Qn>, <Qm>
\]

if \(!\text{HaveQRDMLAHExt}()\) then UNDEFINED;
if \(\text{InITBlock}()\) then UNPREDICTABLE;
if \(Q == '1' \&\& (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1')\) then UNDEFINED;
if \(\text{size} == '00' \| \text{size} == '11'\) then UNDEFINED;
\(\text{add} = \text{TRUE}; \text{scalar} _{\text{form}} = \text{FALSE};\) \(\text{esize} = 8 \ll \text{UInt}(\text{size});\) \(\text{elements} = 64 \div \text{esize};\)
\(d = \text{UInt}(D:\text{Vd}); n = \text{UInt}(N:\text{Vn}); m = \text{UInt}(M:\text{Vm});\) \(\text{regs} = \text{if } Q == '0' \text{ then } 1 \text{ else } 2;\)

**CONSTRAINED UNPREDICTABLE behavior**

If \(\text{InITBlock}()\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T2 (Armv8.1)**

\[
\begin{array}{cccccccccccccc}
1 & 1 & 1 & 0 & D & \text{size} & Vn & Vd & 1 & 0 & 1 & 1 & N & Q & M & 1 & Vm
\end{array}
\]

64-bit SIMD vector \((Q == 0)\)

\[
\text{VQRDMLAH}\{<q>\}.<dt> <Dd>, <Dn>, <Dm[x]>\]

128-bit SIMD vector \((Q == 1)\)

\[
\text{VQRDMLAH}\{<q>\}.<dt> <Qd>, <Qn>, <Qm[x]>\]

if \(!\text{HaveQRDMLAHExt}()\) then UNDEFINED;
if \(\text{InITBlock}()\) then UNPREDICTABLE;
if \(\text{size} == '11'\) then SEE "Related encodings";
if \(\text{size} == '00'\) then UNDEFINED;
if \(Q == '1' \&\& (Vd<0> == '1' || Vn<0> == '1')\) then UNDEFINED;
\(\text{add} = \text{TRUE}; \text{scalar} _{\text{form}} = \text{FALSE}; \text{esize} = 16; \text{elements} = 4;\)
\(m = \text{UInt}(Vm<2:0>); \text{index} = \text{UInt}(M:Vm<3>);\)
\(\text{size} == '10'\) then \(\text{esize} = 32; \) elements = 2; \(m = \text{UInt}(Vm); \text{index} = \text{UInt}(M);\)

**CONSTRAINED UNPREDICTABLE behavior**

If \(\text{InITBlock}()\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.
Assembler Symbols

<q> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the operands, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations();  CheckAdvSIMDEnabled();
round_const = 1 << (esize-1);
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = SInt(Elem[D[n+r],e,esize]);
    op3 = SInt(Elem[D[d+r],e,esize]) << esize;
    if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
    (result, sat) = SignedSatQ((op3 + 2*(op1*op2) + round_const) >> esize, esize);
    Elem[D[d+r],e,esize] = result;
    if sat then FPSCR.QC = '1';
VQRDMLSH

Vector Saturating Rounding Doubling Multiply Subtract Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, \textit{FPSCR}.QC, is set if saturation occurs. For details see \textit{Pseudocode details of saturation}.

Depending on settings in the \textit{CPACR}, \textit{NSACR}, and \textit{HCPTR} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be \textit{UNDEFINED}, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support}.

It has encodings from the following instruction sets: A32 (\textit{A1} and \textit{A2}) and T32 (\textit{T1} and \textit{T2}).

\textbf{A1 (Armv8.1)}

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

64-bit SIMD vector (\(Q == 0\))

VQRDMLSH\{<q>\}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (\(Q == 1\))

VQRDMLSH\{<q>\}.<dt> <Qd>, <Qn>, <Qm>

if !\textit{HaveQRDMLAHExt}() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = FALSE; scalar_form = FALSE; esize = 8 << \textit{UInt}(size); elements = 64 DIV esize;
d = \textit{UInt}(D:Vd); n = \textit{UInt}(N:Vn); m = \textit{UInt}(M:Vm); regs = if Q == '0' then 1 else 2;

\textbf{A2 (Armv8.1)}

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

64-bit SIMD vector (\(Q == 0\))

VQRDMLSH\{<q>\}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (\(Q == 1\))

VQRDMLSH\{<q>\}.<dt> <Qd>, <Qn>, <Dm[x]>

if !\textit{HaveQRDMLAHExt}() then UNDEFINED;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = FALSE; scalar_form = TRUE; d = \textit{UInt}(D:Vd); n = \textit{UInt}(N:Vn); regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16; elements = 4; m = \textit{UInt}(Vm<2:0>); index = \textit{UInt}(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = \textit{UInt}(Vm); index = \textit{UInt}(M);

\textbf{T1 (Armv8.1)}
64-bit SIMD vector (Q == 0)

VQRDMLSH\{<q>\}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VQRDMLSH\{<q>\}.<dt> <Qd>, <Qn>, <Qm>

if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == ‘1’ && (Vd<0> == ‘1’ || Vn<0> == ‘1’ || Vm<0> == ‘1’) then UNDEFINED;
if size == ‘00’ || size == ‘11’ then UNDEFINED;
add = FALSE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == ‘0’ then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2
(Armv8.1)

64-bit SIMD vector (Q == 0)

VQRDMLSH\{<q>\}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VQRDMLSH\{<q>\}.<dt> <Qd>, <Qn>, <Qm[x]>

if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if size == ‘11’ then SEE “Related encodings”;
if size == ‘00’ then UNDEFINED;
if Q == ‘1’ && (Vd<0> == ‘1’ || Vn<0> == ‘1’) then UNDEFINED;
add = FALSE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if size == ‘01’ then esize = 16; elements = 4; index = UInt(M:<Vm<<3>>);
if size == ‘10’ then esize = 32; elements = 2; index = UInt(M);

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.
**Assembler Symbols**

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3">. If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```
EncodingSpecificOperations(); CheckAdvSIMEnabled();
round_const = 1 << (esize-1);
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = SInt(Elem[D[n+r],e,esize]);
    op3 = SInt(Elem[D[d+r],e,esize]) << esize;
    if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
    (result, sat) = SignedSatQ((op3 - 2*(op1*op2) + round_const) >> esize, esize);
    Elem[D[d+r],e,esize] = result;
    if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VQRDMULH

Vector Saturating Rounding Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are rounded. For truncated results see VQDMULH.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

64-bit SIMD vector (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>,} {<Dn>}, <Dm>

128-bit SIMD vector (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>,} {<Qn>}, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

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

size

64-bit SIMD vector (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>}, {<Dn>}, <Dm[x]>

128-bit SIMD vector (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>}, {<Qn>}, <Dm[x]>

if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); index = UInt(Vm<2:0>);
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
64-bit SIMD vector (Q == 0)

\[ \text{VQRDMULH}\{c\}\{q\}\<d>, \<Dd>, \<Dn>, \<Dm> \]

128-bit SIMD vector (Q == 1)

\[ \text{VQRDMULH}\{c\}\{q\}\<d>, \<Qd>, \<Qn>, \<Qm> \]

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\text{D} &=& 11 \\
\text{Vn} & | & \text{Vd} & | & \text{Vm} & | & \text{N} & | & \text{M} & | & 0 & | & \text{Vq} \\
\end{array}
\]

size

64-bit SIMD vector (Q == 0)

\[ \text{VQRDMULH}\{c\}\{q\}\<d>, \<Dd>, \<Dn>, \<Dm> \]

128-bit SIMD vector (Q == 1)

\[ \text{VQRDMULH}\{c\}\{q\}\<d>, \<Qd>, \<Qn>, \<Qm> \]

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\text{D} &=& 11 \\
\text{Vn} & | & \text{Vd} & | & \text{Vm} & | & \text{N} & | & \text{M} & | & 0 & | & \text{Vq} \\
\end{array}
\]

size

Assembler Symbols

\(<c>\) For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.

\(<q>\) See Standard assembler syntax fields.

\(<d>\) Is the data type for the elements of the operands, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>S16</td>
</tr>
<tr>
<td>10</td>
<td>S32</td>
</tr>
</tbody>
</table>

\(<Qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

\(<Qn>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

\(<Qm>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Dm[x]>\) Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = SInt(Elem[D[n+r],e,esize]);
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            (result, sat) = SignedSatQ((2*op1*op2 + round_const) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see VQSHL (register).

The first operand and result elements are the same data type, and can be any one of:
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VQRSHL{<c>}{<q>}.<dt> {<Dd>}, {<Dm>, <Dn>}

128-bit SIMD vector (Q == 1)

VQRSHL{<c>}{<q>}.<dt> {<Qd>}, {<Qm>, <Qn>}

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>U</td>
<td>D</td>
<td>size</td>
<td></td>
<td>Vn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VQRSHL{<c>}{<q>}.<dt> {<Dd>}, {<Dm>, <Dn>}

128-bit SIMD vector (Q == 1)

VQRSHL{<c>}{<q>}.<dt> {<Qd>}, {<Qm>, <Qn>}

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.
Is the data type for the elements of the vectors, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U size</th>
<th>dt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>0 11</td>
<td>S64</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
<tr>
<td>1 11</td>
<td>U64</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D][n+r],e,esize)<7:0>;
            round_const = 1 << (-1-shift); // 0 for left shift, 2^(n-1) for right shift
            operand = Int(Elem[D][m+r],e,esize], unsigned);
            (result, sat) = SatQ((operand + round_const) << shift, esize, unsigned);
            Elem[D][d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the rounded results in a doubleword vector.

For truncated results, see VQSHRN and VQSHRUN.

The operand elements must all be the same size, and can be any one of:
- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

For truncated results, see VQSHRN and VQSHRUN.

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 1  | U  | 1  | D  | imm6 | Vd  | 1  | 0  | 0  | op | 0  | 1  | M  | 1  | Vm  |

Signed result (U == 000xxx) & op == 1)

VQRSHRN<c>}{q}.<type><size> <Dd>, <Qm>, #$imm

Unsigned result (U == 1 & & (imm6 == 000xxx) & & op == 0)

VQRSHRUN<c>}{q}.<type><size> <Dd>, <Qm>, #$imm

if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' & & op == '0' then SEE "VRSRHN";
if Vm<0> == '1' then UNDEFINED;

if imm6 == '001xxx' then:
esize = 8;
Elements = 8;
shift_amount = 16 - Uint(imm6);

if imm6 == '01xxxx' then:
esize = 16;
Elements = 4;
shift_amount = 32 - Uint(imm6);

if imm6 == '1xxxxx' then:
esize = 32;
Elements = 2;
shift_amount = 64 - Uint(imm6);

src_unsigned = (U == '1' & & op == '1');
dest_unsigned = (U == '1');

if U == '0' & & op == '0' then:
d = Uint(D:Vd);
m = Uint(M:Vm);

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | D  | imm6 | Vd  | 1  | 0  | 0  | op | 0  | 1  | M  | 1  | Vm  |
Signed result (!(imm6 == 000xxx) & op == 1)

VQRSHRN{<c><q>}.<type><size> <Dd>, <Qm>, #imm

Unsigned result (U == 1 & !(imm6 == 000xxx) & op == 0)

VQRSHRUN{<c><q>}.<type><size> <Dd>, <Qm>, #imm

if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' & op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<type> For the signed result variant: is the data type for the elements of the vectors, encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>&lt;type&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
</tr>
</tbody>
</table>

For the unsigned result variant: is the data type for the elements of the vectors, encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>&lt;type&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
</tr>
</tbody>
</table>

<size> Is the data size for the elements of the vectors, encoded in "imm6<5:3>":

<table>
<thead>
<tr>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>16</td>
</tr>
<tr>
<td>01x</td>
<td>32</td>
</tr>
<tr>
<td>1xx</td>
<td>64</td>
</tr>
</tbody>
</table>

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  round_const = 1 << (shift_amount - 1);
  for e = 0 to elements-1
    operand = Int(Elem[Qin][m>>1],e,2*esize], src_unsigned);
    (result, sat) = SatQ((operand + round_const) >> shift_amount, esize, dest_unsigned);
    Elem[D[d],e,esize] = result;
    if sat then FPSCR.QC = '1';
VQRSHRN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed rounded results in a doubleword vector.

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

Signed result

VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 1 1 1 1 1 1 | D | 1 1 | size | 1 0 | Vd | 0 0 1 0 1 x | M | 0 | Vm
```

Signed result

VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

- `<c>` For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
  
  For encoding T1: see Standard assembler syntax fields.

- `<q>` See Standard assembler syntax fields.

- `<dt>` Is the data type for the elements of the operand, encoded in "op<0>:size":

<table>
<thead>
<tr>
<th>op&lt;0&gt;</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>S16</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>S32</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S64</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>U16</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>U32</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>U64</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.`
**Operation**

The description of **VOMOVN, VOMOVUN** gives the operational pseudocode for this instruction.
VQRSHRUN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned rounded results in a doubleword vector.

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

Unsigned result

VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | D | 1  | 1  | size | 1  | 0  | Vd | 0  | 0  | 1  | 0  | 0  | 1  | M | 0 | Vm |

Unsigned result

VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

- `<c>` For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1: see Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<dt>` Is the data type for the elements of the operand, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>S16</td>
</tr>
<tr>
<td>01</td>
<td>S32</td>
</tr>
<tr>
<td>10</td>
<td>S64</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.

Operation

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.
Vector Saturating Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in a second vector.

The operand elements must all be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are the same size as the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, \texttt{FPSCR.QC}, is set if saturation occurs.

For details see \textit{Pseudocode details of saturation}.

Depending on settings in the \textit{CPACR}, \textit{NSACR}, and \textit{HCPTR} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be \texttt{UNDEFINED}, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support}.

It has encodings from the following instruction sets: A32 (\texttt{A1}) and T32 (\texttt{T1}).

\texttt{A1}

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 U 1 D  imm6   Vd   0 1 1 op L Q M 1     Vm
\end{verbatim}

\texttt{VQSHL,double,signed-result (!!(imm6 == 000xxx && L == 0) && op == 1 && Q == 0)}

\texttt{VQSHL{<c>}{<q>}.<type><size> {<Dd>},} \texttt{<Dm>, #<imm>}

\texttt{VQSHL,quad,signed-result (!!(imm6 == 000xxx && L == 0) && op == 1 && Q == 1)}

\texttt{VQSHL{<c>}{<q>}.<type><size> {<Qd>},} \texttt{<Qm>, #<imm>}

\texttt{VQSHLU,double,unsigned-result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0)}

\texttt{VQSHLU{<c>}{<q>}.<type><size> {<Dd>},} \texttt{<Dm>, #<imm>}

\texttt{VQSHLU,quad,unsigned-result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1)}

\texttt{VQSHLU{<c>}{<q>}.<type><size> {<Qd>},} \texttt{<Qm>, #<imm>}

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if U == '0' \&& op == '0' then UNDEFINED;
if Q == '1' \&& (Vd<0> == '1' \|\| Vm<0> == '1') then UNDEFINED;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '001xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '01xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = UInt(imm6);
src_unsigned = (U == '1' \&& op == '1'); dest_unsigned = (U == '1');
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

\texttt{T1}

\begin{verbatim}
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 U 1 1 1 1 1 1 D  imm6   Vd   0 1 1 op L Q M 1     Vm
\end{verbatim}
VQSHL, double, signed-result $(!(\text{imm6} == 000xxx \&\& L == 0) \&\& op == 1 \&\& Q == 0)$

\[
\text{VQSHL}\{\text{<c>}}\{\text{<q>}}\}.\text{<type>size } \{\text{<Dd>},} \\text{<Dm>}, \#\text{<imm>}
\]

VQSHL, quad, signed-result $(!(\text{imm6} == 000xxx \&\& L == 0) \&\& op == 1 \&\& Q == 1)$

\[
\text{VQSHL}\{\text{<c>}}\{\text{<q>}}\}.\text{<type>size } \{\text{<Qd>},} \\text{<Qm>}, \#\text{<imm>}
\]

VQSHLU, double, unsigned-result $(U == 1 \&\& !(\text{imm6} == 000xxx \&\& L == 0) \&\& op == 0 \&\& Q == 0)$

\[
\text{VQSHLU}\{\text{<c>}}\{\text{<q>}}\}.\text{<type>size } \{\text{<Dd>},} \\text{<Dm>}, \#\text{<imm>}
\]

VQSHLU, quad, unsigned-result $(U == 1 \&\& !(\text{imm6} == 000xxx \&\& L == 0) \&\& op == 0 \&\& Q == 1)$

\[
\text{VQSHLU}\{\text{<c>}}\{\text{<q>}}\}.\text{<type>size } \{\text{<Qd>},} \\text{<Qm>}, \#\text{<imm>}
\]

\[
\text{if } (L:\text{imm6}) == '0000xxx' \text{ then SEE "Related encodings";}
\text{if } U == '0' \&\& op == '0' \text{ then UNDEFINED;}
\text{if } Q == '1' \&\& (Vd<0> == '1' \text{ || Vm<0> == '1'}') \text{ then UNDEFINED;}
\text{case } L:\text{imm6} \text{ of}
\begin{align*}
\text{when } '0001xxx' & \text{ esize } = 8; \text{ elements } = 8; \text{ shift_amount } = \text{UInt}(\text{imm6}) - 8; \\
\text{when } '00lxxxxx' & \text{ esize } = 16; \text{ elements } = 4; \text{ shift_amount } = \text{UInt}(\text{imm6}) - 16; \\
\text{when } '0lxxxxxx' & \text{ esize } = 32; \text{ elements } = 2; \text{ shift_amount } = \text{UInt}(\text{imm6}) - 32; \\
\text{when } '1xxxxxx' & \text{ esize } = 64; \text{ elements } = 1; \text{ shift_amount } = \text{UInt}(\text{imm6});
\end{align*}
\text{src unsigned } = (U == '1' \&\& op == '1'); \text{ dest unsigned } = (U == '1');
\text{d } = \text{UInt}(D:\text{Vd}); \text{ m } = \text{UInt}(M:\text{Vm}); \text{ regs } = \text{if } Q == '0' \text{ then 1 else 2;}
\]

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

**Assembler Symbols**

- **<c>** For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
  
  For encoding T1: see Standard assembler syntax fields.

- **<q>** See Standard assembler syntax fields.

- **<type>** Is the data type for the elements of the vectors, encoded in “U”:

<table>
<thead>
<tr>
<th>U</th>
<th>&lt;type&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
</tr>
</tbody>
</table>

- **<size>** Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

<table>
<thead>
<tr>
<th>L</th>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>1xx</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>xxx</td>
<td>64</td>
</tr>
</tbody>
</table>

- **<Qd>** Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

- **<Qm>** Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- **<Dm>** Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

- **<imm>** Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.
Operation

if `ConditionPassed()` then
  EncodingSpecificOperations(); `CheckAdvSIMDEnabled()`;
  for `r` = 0 to `regs-1`
    for `e` = 0 to `elements-1`
      operand = `Int(Elem[D][m+r], e, esize], src_unsigned);
      (result, sat) = `SatQ(operand << shift_amount, esize, dest_unsigned);
      `Elem[D][d+r], e, esize] = result;
      if sat then `FPSCR.QC = '1';

**VQSHL (register)**

Vector Saturating Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results are truncated. For rounded results, see [VQRSHL](#).

The first operand and result elements are the same data type, and can be any one of:
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, **FPSCR** QC, is set if saturation occurs. For details see *Pseudocode details of saturation.*

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support.*

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

**64-bit SIMD vector (Q == 0)**

```
VQSHL{<c>}<{q}>.{<dt>} {<Dd>,} <Dm>, <Dn>
```

### 128-bit SIMD vector (Q == 1)

```
VQSHL{<c>}<{q}>.{<dt>} {<Qd>,} <Qm>, <Qn>
```

if Q == '1' & (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;

unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**64-bit SIMD vector (Q == 0)**

```
VQSHL{<c>}<{q}>.{<dt>} {<Dd>,} <Dm>, <Dn>
```

### 128-bit SIMD vector (Q == 1)

```
VQSHL{<c>}<{q}>.{<dt>} {<Qd>,} <Qm>, <Qn>
```

if Q == '1' & (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;

unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;

**Assembler Symbols**

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.
  - For encoding T1: see [Standard assembler syntax fields](#).

- <q> See [Standard assembler syntax fields](#).
Is the data type for the elements of the vectors, encoded in "U:size":

<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>0 11</td>
<td>S64</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
<tr>
<td>1 11</td>
<td>U64</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D][n+r],e,esize)<7:0>;
            operand = Int(Elem[D][m+r],e,esize], unsigned);
            (result,sat) = SatQ(operand << shift, esize, unsigned);
            Elem[D][d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VQSHRN, VQSHRUN

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the truncated results in a doubleword vector. For rounded results, see VQRSHRN and VQRSHRUN.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

Signed result (!(!imm6 == 000xxx) && op == 1)

VQSHRN{<c>}{<q>},<type><size> <Dd>, <Qm>, #<imm>

Unsigned result (U == 1 && !(!imm6 == 000xxx) && op == 0)

VQSHRUN{<c>}{<q>},<type><size> <Dd>, <Qm>, #<imm>

```plaintext
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
    case imm6 of
        when '001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
        when '01xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
        when '1xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
    src_unsigned = (U == '1' && op == '1'); dest_unsigned = (U == '1');
    d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
</tr>
</tbody>
</table>
Signed result (!\text{imm6} = 000xxx) \& \ op = 1)

\text{VQSHRN\{<c>}\{<q>\}.<\text{type}><\text{size}> <\text{Dd}>, <\text{Qm}>, #<\text{imm}>}

Unsigned result (U = 1 \& !\text{imm6} = 000xxx) \& \ op = 0)

\text{VQSHRUN\{<c>}\{<q>\}.<\text{type}><\text{size}> <\text{Dd}>, <\text{Qm}>, #<\text{imm}>}

\begin{verbatim}
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' \& op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
\text{case imm6 of}
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - \text{UInt}(imm6);
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - \text{UInt}(imm6);
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - \text{UInt}(imm6);
\text{src_unsigned} = (U == '1' \& op == '1');  \text{dest_unsigned} = (U == '1');
d = \text{UInt}(D:Vd);  m = \text{UInt}(M:Vm);
\end{verbatim}

Related encodings: See \textit{Advanced SIMD one register and modified immediate} for the T32 instruction set, or \textit{Advanced SIMD one register and modified immediate} for the A32 instruction set.

\textbf{Assembler Symbols}

\begin{itemize}
  \item \texttt{<c>} \quad \text{For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.}
  \item \texttt{<q>} \quad \text{See Standard assembler syntax fields.}
  \item \texttt{<type>} \quad \text{For the signed result variant: is the data type for the elements of the vectors, encoded in "U":}
\end{itemize}

\begin{tabular}{|c|c|}
\hline
\textbf{U} & \textbf{<type>} \\
\hline
0 & S \\
1 & U \\
\hline
\end{tabular}

\begin{itemize}
  \item \texttt{<size>} \quad \text{Is the data size for the elements of the vectors, encoded in "imm6<5:3>":}
\end{itemize}

\begin{tabular}{|c|c|}
\hline
\textbf{imm6<5:3>} & \textbf{<size>} \\
\hline
001 & 16 \\
01x & 32 \\
1xx & 64 \\
\hline
\end{tabular}

\begin{itemize}
  \item \texttt{<Dd>} \quad \text{Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.}
  \item \texttt{<Qm>} \quad \text{Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.}
  \item \texttt{<imm>} \quad \text{Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.}
\end{itemize}

\textbf{Operation}

\begin{verbatim}
if \text{ConditionPassed()} then
    \text{EncodingSpecificOperations(); CheckAdvSIMDEnabled();}
    \text{for e = 0 to elements-1}
        \text{operand = \text{Int}(Elem[Qin[mm]>1],e,2*esize], src_unsigned);
        (result, sat) = \text{SatQ}(operand >> \text{shift_amount}, esize, dest_unsigned);
        \text{Elem[D][e,e,esize] = result;}
        if sat then FPSCR.QC = '1';
\end{verbatim}
VQSHRN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed truncated results in a doubleword vector.

This is a pseudo-instruction of \texttt{VQMOVN, VQMOVUN}. This means:

- The encodings in this description are named to match the encodings of \texttt{VQMOVN, VQMOVUN}.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of \texttt{VQMOVN, VQMOVUN} gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (\texttt{A1}) and T32 (\texttt{T1}).

\begin{verbatim}
A1

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

Signed result

\texttt{VQSHRN\{<c>\}{<q>}.<dt> <Dd>, <Qm>, #0}

is equivalent to

\texttt{VQMOVN\{<c>\}{<q>}.<dt> <Dd>, <Qm>}

\begin{verbatim}
T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>1</td>
<td>0</td>
<td>Vd</td>
</tr>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Signed result

\texttt{VQSHRN\{<c>\}{<q>}.<dt> <Dd>, <Qm>, #0}

is equivalent to

\texttt{VQMOVN\{<c>\}{<q>}.<dt> <Dd>, <Qm>}

\end{verbatim}

\textbf{Assembler Symbols}

\begin{verbatim}
<\texttt{c}> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<\texttt{q}> See Standard assembler syntax fields.

<\texttt{dt}> Is the data type for the elements of the operand, encoded in "op<0>:size":

\begin{tabular}{|c|c|c|}
\hline
op<0> & size & \texttt{dt} \\
\hline
0    & 00   & S16  \\
0    & 01   & S32  \\
0    & 10   & S64  \\
0    & 11   & RESERVED \\
1    & 00   & U16  \\
1    & 01   & U32  \\
1    & 10   & U64  \\
1    & 11   & RESERVED \\
\hline
\end{tabular}

<\texttt{Dd}> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<\texttt{Qm}> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <\texttt{Qm}>*2.
\end{verbatim}
Operation

The description of \texttt{VOMOVN, VOMOVUN} gives the operational pseudocode for this instruction.
VQSHRUN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned truncated results in a doubleword vector.

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

\[
\begin{array}{cccccccccccccccccc}
\text{1} & \text{1} & \text{1} & \text{1} & \text{0} & \text{1} & \text{1} & \text{1} & \text{D} & \text{1} & \text{1} & \text{size} & \text{1} & \text{0} & \text{Vd} & \text{0} & \text{0} & \text{1} & \text{0} & \text{1} & \text{M} & \text{0} & \text{Vm} & \text{op}
\end{array}
\]

**Unsigned result**

\[
\text{VQSHRUN}\{<c>\}{<q>}.<\text{dt}> <\text{Dd}>, <\text{Qm}>, \#0
\]

is equivalent to

\[
\text{VQMOVUN}\{<c>\}{<q>}.<\text{dt}> <\text{Dd}>, <\text{Qm}>
\]

**T1**

\[
\begin{array}{cccccccccccccccccc}
\text{1} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} & \text{1} & \text{D} & \text{1} & \text{1} & \text{size} & \text{1} & \text{0} & \text{Vd} & \text{0} & \text{0} & \text{1} & \text{0} & \text{1} & \text{M} & \text{0} & \text{Vm} & \text{op}
\end{array}
\]

**Unsigned result**

\[
\text{VQSHRUN}\{<c>\}{<q>}.<\text{dt}> <\text{Dd}>, <\text{Qm}>, \#0
\]

is equivalent to

\[
\text{VQMOVUN}\{<c>\}{<q>}.<\text{dt}> <\text{Dd}>, <\text{Qm}>
\]

**Assembler Symbols**

- **<c>** For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1: see Standard assembler syntax fields.
- **<q>** See Standard assembler syntax fields.
- **<dt>** Is the data type for the elements of the operand, encoded in "size":

\[
\begin{array}{c|c}
\text{size} & <\text{dt}> \\
\hline
00 & S16 \\
01 & S32 \\
10 & S64 \\
11 & RESERVED
\end{array}
\]

- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- **<Qm>** Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation**

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.
VQSUB

Vector Saturating Subtract subtracts the elements of the second operand vector from the corresponding elements of the first operand vector, and places the results in the destination vector. Signed and unsigned operations are distinct. The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode details of saturation. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 | U | 0 | D | size | Vn | Vd | 0 0 1 0 | N | Q | M | 1 | Vm
```

64-bit SIMD vector (Q == 0)

VQSUB{<c>}{<q>}{<dt>}{<Dd>,} <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VQSUB{<c>}{<q>}{<dt>}{<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 | U | 1 1 1 1 0 | D | size | Vn | Vd | 0 0 1 0 | N | Q | M | 1 | Vm
```

64-bit SIMD vector (Q == 0)

VQSUB{<c>}{<q>}{<dt>}{<Dd>,} <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VQSUB{<c>}{<q>}{<dt>}{<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<
For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<
See Standard assembler syntax fields.

<dt>
Is the data type for the elements of the vectors, encoded in “U:size”:
<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>0 11</td>
<td>S64</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
<tr>
<td>1 11</td>
<td>U64</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            diff = Int(Elem[D][n+r],e,esize], unsigned) - Int(Elem[D][m+r],e,esize], unsigned);
            (Elem[D][d+r],e,esize], sat) = SatQ(diff, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VRADDHN

Vector Rounding Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are rounded. For truncated results, see VADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

VRADDHN{<c>}{<q>}{<dt>}{<Dd>, <Qn>, <Qm>}

if size == '11' then SEE “Related encodings”;
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

VRADDHN{<c>}{<q>}{<dt>}{<Dd>, <Qn>, <Qm>}

if size == '11' then SEE “Related encodings”;
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operands, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>116</td>
</tr>
<tr>
<td>01</td>
<td>132</td>
</tr>
<tr>
<td>10</td>
<td>164</td>
</tr>
</tbody>
</table>

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

- The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VRECPE

Vector Reciprocal Estimate finds an approximate reciprocal of each element in the operand vector, and places the results in the destination vector.

The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see Floating-point reciprocal square root estimate and step.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

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

64-bit SIMD vector (Q == 0)

VRECPE{<c>}{<q>}.${dt} <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRECPE{<c>}{<q>}.${dt} <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
floating_point = (F == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

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

64-bit SIMD vector (Q == 0)

VRECPE{<c>}{<q>}.${dt} <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRECPE{<c>}{<q>}.${dt} <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONstrained UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:
  • The instruction is UNDEFINED.
  • The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler Symbols**

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional. For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in “F:size”:

<table>
<thead>
<tr>
<th>F</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>U32</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Newton-Raphson iteration**

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see Floating-point reciprocal estimate and step.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[0+r],e,esize] = FPRecipEstimate(Elem[D[0+r],e,esize], StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = UnsignedRecipEstimate(Elem[D[m+r],e,esize]);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Vector Reciprocal Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 2.0, and places the results into the elements of the destination vector. The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see Floating-point reciprocal estimate and step.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

64-bit SIMD vector (Q == 0)

VRECPS{<c>{<q>}.dt} {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VRECPS{<c>{<q>}.dt} {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VRECPS{<c>{<q>}.dt} {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VRECPS{<c>{<q>}.dt} {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

### CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration
For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see Floating-point reciprocal estimate and step.

Operation

if ConditionPassed() then

    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRecipStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
VREV16

Vector Reverse in halfwords reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector. There is no distinction between data types, other than size. The following figure shows an example of the operation of VREV16 doubleword operation.

VREV16.8, doubleword

![](image)

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

![A1 encoding](image)

64-bit SIMD vector (Q == 0)

VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>

if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

![T1 encoding](image)
64-bit SIMD vector (Q == 0)

\[ \text{VREV16}\{\langle c\rangle}\{\langle q\rangle}\}.\langle dt\rangle <\langle D\rangle, <\langle Dm\rangle> \]

128-bit SIMD vector (Q == 1)

\[ \text{VREV16}\{\langle c\rangle}\{\langle q\rangle}\}.\langle dt\rangle <\langle Q\rangle, <\langle Qm\rangle> \]

if \( \text{UInt}(op) + \text{UInt}(size) \geq 3 \) then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 & UInt(size);
integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operand, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();

bits(64) result;
integer element;
integer rev_element;
for r = 0 to regs-1
  element = 0;
  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[D[m+r], element, esize];
      element = element + 1;
      rev_element = rev_element - 1;
  D[d+r] = result;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
VREV32

Vector Reverse in words reverses the order of 8-bit or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows an example of the operation of VREV32 doubleword operations.

Depending on settings in the \textit{CPACR}, \textit{NSACR}, and \textit{HCPTR} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support}.

It has encodings from the following instruction sets: A32 (\textit{A1}) and T32 (\textit{T1}).

A1

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 1 1 | D | 1 | size | 0 0 | Vd | 0 0 0 0 1 | Q | M | 0 | Vm
\end{verbatim}

64-bit SIMD vector (Q == 0)

VREV32\{\langle c\rangle\}{\langle q\rangle}.\langle dt\rangle <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VREV32\{\langle c\rangle\}{\langle q\rangle}.\langle dt\rangle <Qd>, <Qm>

\begin{verbatim}
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
\end{verbatim}

T1

\begin{verbatim}
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 | D | 1 | size | 0 0 | Vd | 0 0 0 0 1 | Q | M | 0 | Vm
\end{verbatim}
64-bit SIMD vector (Q == 0)

VREV32{{c}\{q}\}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VREV32{{c}\{q}\}.<dt> <Qd>, <Qm>

if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
integer contains = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operand, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>1x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();

  bits(64) result;
  integer element;
  integer rev_element;
  for r = 0 to regs-1
    element = 0;
    for c = 0 to contains-1
      rev_element = element + elements_per_container - 1;
      for e = 0 to elements_per_container-1
        $\text{Elem}[\text{result, rev_element, esize}] = \text{Elem}[\text{D[m+r], element, esize}]$
        element = element + 1;
        rev_element = rev_element - 1;
      $\text{D}[d+r] = \text{result}$;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**VREV64**

Vector Reverse in doublewords reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector. There is no distinction between data types, other than size. The following figure shows an example of the operation of VREV64 doubleword operations.

![VREV64 Example](image)

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 30 29 28 27 26 25 24 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 | 1 | 1 | D | 1 | 1 | size | 0 | 0 | Vd | 0 | 0 | 0 | 0 | Q | M | 0 | Vm |

**64-bit SIMD vector (Q == 0)**

VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>

**128-bit SIMD vector (Q == 1)**

VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>

```plaintext
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | size | 0 | 0 | Vd | 0 | 0 | 0 | 0 | Q | M | 0 | Vm |

op
64-bit SIMD vector (Q == 0)
VREV64\{<c}\{<q>\}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)
VREV64\{<c}\{<q>\}.<dt> <Qd>, <Qm>
if UInt(op) + UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
    case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
    integer containers = 64 DIV container_size;
    integer elements_per_container = container_size DIV esize;
    d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler Symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
    For encoding T1: see Standard assembler syntax fields.
\(<q>\) See Standard assembler syntax fields.
\(<dt>\) Is the data type for the elements of the operand, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
\(<Qm>\) Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
\(<Dm>\) Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    bits(64) result;
    integer element;
    integer rev_element;
    for r = 0 to regs-1
        element = 0;
        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[D[m+r], element, esize];
                element = element + 1;
                rev_element = rev_element - 1;
        D[d+r] = result;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
• The values of the NZCV flags.

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

Vector Rounding Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The results of the halving operations are rounded. For truncated results, see VHADD. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VRHADD{<c>}{<q>}.<dt> {<Dd>,}<{Dn>,}<{Dm}>

128-bit SIMD vector (Q == 1)

VRHADD{<c>}{<q>}.<dt> {<Qd>,}<{Qn>,}<{Qm}>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 31 30 29 28 27 26 25 24 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 | U | 1 1 1 1 0 | D | size | Vn | Vd | 0 0 0 1 | N | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VRHADD{<c>}{<q>}.<dt> {<Dd>,}<{Dn>,}<{Dm}>

128-bit SIMD vector (Q == 1)

VRHADD{<c>}{<q>}.<dt> {<Qd>,}<{Qn>,}<{Qm}>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.
;q> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the operands, encoded in “U:size”:
<dt> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<tn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<tm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<dt> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = op1 + op2 + 1;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Round floating-point to integer towards Nearest with Ties to Away rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

#### 64-bit SIMD vector (Q == 0)

VRINTA{<q>}.<dt> <Dd>, <Dm>

#### 128-bit SIMD vector (Q == 1)

VRINTA{<q>}.<dt> <Qd>, <Qm>

if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;

#### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 1 1 1 1 | D | 1 | 1 | size | 1 | 0 | Vd | 0 | 1 | 0 | 1 | 0 | Q | M | 0 | Vm |
|---------------------------------|-----------------|------|----|-----|-----|---|----|-----|-----|----|----|---|----|-----|----|---|----|-----|----|---|----|-----|----|---|----|-----|----|---|

#### 64-bit SIMD vector (Q == 0)

VRINTA{<q>}.<dt> <Dd>, <Dm>

#### 128-bit SIMD vector (Q == 1)

VRINTA{<q>}.<dt> <Qd>, <Qm>

if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;

#### CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD two registers misc for the T32 instruction set, or Advanced SIMD two registers misc for the A32 instruction set.

Assembler Symbols

<q>  See Standard assembler syntax fields.
<dt>  Is the data type for the elements of the vectors, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations();  CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
VRINTA (floating-point)

Round floating-point to integer to Nearest with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

Half-precision scalar (size == 01)
(Armv8.2)

VRINTA{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTA{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTA{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

T1
Half-precision scalar (size == 01)
(Armv8.2)

VRINTA{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTA{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTA{<q>}.F64 <Dd>, <Dm>

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q> See Standard assembler syntax fields.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m] <15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
VRINTM (Advanced SIMD)

Vector Round floating-point to integer towards -Infinity rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards -Infinity rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VRINTM{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTM{<q>}.<dt> <Qd>, <Qm>

if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
  d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| 1 1 1 1 1 1 1 1 1 |
| D | 1 | 1 | size | 1 | 0 | Vd | 0 | 1 | 1 | 0 | 1 | Q | M | 0 | Vm |
| op |

64-bit SIMD vector (Q == 0)

VRINTM{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTM{<q>}.<dt> <Qd>, <Qm>

if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
  d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:
The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD two registers misc for the T32 instruction set, or Advanced SIMD two registers misc for the A32 instruction set.

Assembler Symbols

<q>  See Standard assembler syntax fields.
<dt> Is the data type for the elements of the vectors, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**VRINTM (floating-point)**

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

RM  size

**Half-precision scalar (size == 01)**

(ARMv8.2)

VRINTM{<q>}.F16 <Sd>, <Sm>

**Single-precision scalar (size == 10)**

VRINTM{<q>}.F32 <Sd>, <Sm>

**Double-precision scalar (size == 11)**

VRINTM{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  |

RM  size

**Half-precision scalar (size == 01)**

(ARMv8.2)

VRINTM{<q>}.F16 <Sd>, <Sm>

**Single-precision scalar (size == 10)**

VRINTM{<q>}.F32 <Sd>, <Sm>

**Double-precision scalar (size == 11)**

VRINTM{<q>}.F64 <Dd>, <Dm>

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
CONstrained UNPredictable behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONstrained UNPredictable behavior of this instruction, see Architectural Constraints on UNPredictable behaviors.

Assembler Symbols

<q> See Standard assembler syntax fields.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
when 16
  S[d] = Zeros(16); FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
when 32
  S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
when 64
  D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
VRINTN (Advanced SIMD)

Vector Round floating-point to integer to Nearest rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VRINTN{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTN{<q>}.<dt> <Qd>, <Qm>

if op<2> != op<0> then SEE "Related encodings";
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' & !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
    d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VRINTN{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTN{<q>}.<dt> <Qd>, <Qm>

if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' & !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
    d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:
The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD two registers misc for the T32 instruction set, or Advanced SIMD two registers misc for the A32 instruction set.

### Assembler Symbols

- `<q>` See Standard assembler syntax fields.
- `<dt>` Is the data type for the elements of the vectors, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.

- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

### Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = Elem[D[m+r], e, esize];
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
    Elem[D[d+r], e, esize] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 0  | 1  | D  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | Vd | 1  | 0  | != | 00 | 0  | 1  | M  | 0  | Vm |

RM | size

Half-precision scalar (size == 01) (Armv8.2)

VRINTN{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTN{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTN{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' & & !HaveFP16Ext()) then UNDEFINED; rounding = FPDecodeRM(RM); exact = FALSE;

case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Vd</td>
</tr>
</tbody>
</table>

RM | size

Half-precision scalar (size == 01) (Armv8.2)

VRINTN{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTN{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTN{<q>}.F64 <Dd>, <Dm>

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' & & !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;

case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

- `<q>`  See *Standard assembler syntax fields*.
- `<Sd>`  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>`  Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dd>`  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>`  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```plaintext
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VRINTP (Advanced SIMD)

Vector Round floating-point to integer towards +Infinity rounds a vector of floating-point values to integral floating-point values of the same size using the Round towards +Infinity rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VRINTP{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTP{<q>}.<dt> <Qd>, <Qm>

if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && 'HaveFP16Ext') || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
  d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 1 1 1 1 1 1 1 | D | 1 1 | size | 1 0 | Vd | 0 1 1 1 1 | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VRINTP{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTP{<q>}.<dt> <Qd>, <Qm>

if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && 'HaveFP16Ext') || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
  d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:
The instruction is UNDEFINED.
The instruction executes as if it passes the Condition code check.
The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See Advanced SIMD two registers misc for the T32 instruction set, or Advanced SIMD two registers misc for the A32 instruction set.

Assembler Symbols

<q> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the vectors, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = Elem[D[m+r],e,esize];
    result = FPRoundInt(op1, StandardFPSCRFValue(), rounding, exact);
    Elem[D[d+r],e,esize] = result;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 1  | D  | 1  | 1  | 1  | 0  | 1  | 0  | Vd | 1  | 0  | != | 00 | 0  | 1  | M  | 0  | Vm |

RM size

Half-precision scalar (size == 01)
(Armv8.2)

VRINTP{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTP{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTP{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;

case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 0  | D  | 1  | 1  | 0  | 0  | 1  | 0  | Vd | 1  | 0  | != | 00 | 0  | 1  | M  | 0  | Vm |

RM size

Half-precision scalar (size == 01)
(Armv8.2)

VRINTP{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTP{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTP{<q>}.F64 <Dd>, <Dm>

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;

case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
**CONSTRAINED UNPREDICTABLE behavior**

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<q>` See *Standard assembler syntax fields*.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```c
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VRINTR

Round floating-point to integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in the FPSCR. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision scalar (size == 01)
(Armv8.2)

VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 0 | 1 | 1 | 0 | Vd | 1 | 0 | size | 0 | 1 | M | 0 | Vm |
| op |
Half-precision scalar (size == 01)
(Armv8.2)

VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRUANED UNPREDICTABLE behavior
If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEabled(TRUE);
case size of
  when 16
    S[d] = Zerps(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
VRINTX (Advanced SIMD)

Vector round floating-point to integer inexact rounds a vector of floating-point values to integral floating-point values of the same size, using the Round to Nearest rounding mode, and raises the Inexact exception 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VRINTX<{q}>,<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTX<{q}>,<dt> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_TIEEVEN; exact = TRUE;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VRINTX<{q}>,<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTX<{q}>,<dt> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_TIEEVEN; exact = TRUE;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<q>`: See *Standard assembler syntax fields*.
- `<dt>`: Is the data type for the elements of the vectors, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;dt&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- `<Qm>`: Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.
- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>`: Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```c
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for (r = 0 to regs-1
    for (e = 0 to elements-1
        op1 = Elem[D[d+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VRINTX (floating-point)

Round floating-point to integer inexact rounds a floating-point value to an integral floating-point value of the same size, using the rounding mode specified in the FPSCR, and raises an Inexact exception 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
!= 1111 | 1 1 1 0 1 | D 1 1 0 | 1 1 1 | Vd 1 0 | size 0 1 | M 0 | Vm
cond
```

Half-precision scalar (size == 01) (Armv8.2)

VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
exact = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

T1

```plaintext
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 1 1 0 1 | D 1 1 0 | 1 1 1 | Vd 1 0 | size 0 1 | M 0 | Vm
```

VRINTX (floating-point)  Page 1152
Half-precision scalar (size == 01)
(Armv8.2)

VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
extact = TRUE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  rounding = FPRoundingMode(FPSCR[]);
  case esize of
    when 16
      S[d] = Zerps(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
      S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
      D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
VRINTZ (Advanced SIMD)

Vector round floating-point to integer towards Zero rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards Zero rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VRINTZ{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTZ{<q>}.<dt> <Dq>, <Dm>

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 1 1 1 1 | D | 1 | 1 | size | 1 | 0 | Vd | 0 | 1 | 0 | 1 | 1 | Q | M | 0 | Vm |

64-bit SIMD vector (Q == 0)

VRINTZ{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTZ{<q>}.<dt> <Dq>, <Dm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_ZERO; exact = FALSE;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
  d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

if InITBlock() then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.
Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>F16</td>
</tr>
<tr>
<td>10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

EncodingSpecificOperations(); CheckAdvSIMDEnabled();

for r = 0 to regs-1
  for e = 0 to elements-1
    op1 =Elem[D[m+r],e,esize];
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
    Elem[D[d+r],e,esize] = result;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**VRINTZ (floating-point)**

Round floating-point to integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. 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.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

| != 1111 | 1 1 1 0 1 | D | 1 1 | 0 | 1 1 | 0 | Vd | 1 0 | size | 1 1 | M | 0 | Vm |
|-------|-----------|---|-----|---|-----|---|----|----|-----|----|---|---|---|---|
| cond  | op        |

**Half-precision scalar (size == 01)** (Armv8.2)

VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>

**Single-precision scalar (size == 10)**

VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>

**Double-precision scalar (size == 11)**

VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRoundingZERO else FPRoundingMode(FPSCR());
exact = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

**T1**

| 1 1 1 0 1 1 1 0 1 | D | 1 1 | 0 | 1 1 | 0 | Vd | 1 0 | size | 1 1 | M | 0 | Vm |
|-------------------|---|-----|---|-----|---|----|----|-----|----|---|---|---|---|
| op                |   |     |   |     |   |    |    |     |    |   |   |   |   |
Half-precision scalar (size == 01)  
(Armv8.2)

```
VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>
```

if size == '00' | | (size == '01' & & !HaveFP16Ext()) then UNDEFINED;
if size == '01' & & InITBlock() then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- `<c>` See Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  case esize of
    when 16
      S[d] = Zerps(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
      S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
      D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see VSHL.

The first operand and result elements are the same data type, and can be any one of:
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

64-bit SIMD vector (Q == 0)

VRSHL{<c>}{<q>}{<dt>} {<Dd>,} <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VRSHL{<c>}{<q>}{<dt>} {<Qd>,} <Qm>, <Qn>

if Q == '1' & (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector (Q == 0)

VRSHL{<c>}{<q>}{<dt>} {<Dd>,} <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VRSHL{<c>}{<q>}{<dt>} {<Qd>,} <Qm>, <Qn>

if Q == '1' & (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in “U:size”: 
<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>0 11</td>
<td>S64</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
<tr>
<td>1 11</td>
<td>U64</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D][n+r],e,esize)<7:0>);
            round_const = 1 << (-shift-1); // 0 for left shift, 2^(n-1) for right shift
            result = (Int(Elem[D][m+r],e,esize], unsigned) + round_const) << shift;
            Elem[D][d+r],e,esize] = result<esize-1:0>;
```

**Operational information**

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

Vector Rounding Shift Right takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VSHR.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>imm6</td>
<td>Vd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>L</td>
<td>Q</td>
<td>M</td>
<td>1</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 0)

VRSHR{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>

128-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 1)

VRSHR{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | D  | imm6| Vd | 0  | 0  | 1  | 0  | L  | Q  | M  | 1  | Vm |

64-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 0)

VRSHR{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>

128-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 1)

VRSHR{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.
Assembler Symbols

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

See Standard assembler syntax fields.

Is the data type for the elements of the vectors, encoded in “U”:

<table>
<thead>
<tr>
<th></th>
<th>&lt;type&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
</tr>
</tbody>
</table>

Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

<table>
<thead>
<tr>
<th>L</th>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>1xx</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>xxx</td>
<td>64</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  round_const = 1 << (shift_amount - 1);
  for r = 0 to regs-1
    for e = 0 to elements-1
      result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
      Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Rounding Shift Right copies the contents of one SIMD register to another.

This is a pseudo-instruction of VORR (register). This means:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

64-bit SIMD vector (Q == 0)

VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>

128-bit SIMD vector (Q == 1)

VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>

T1

64-bit SIMD vector (Q == 0)

VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>

128-bit SIMD vector (Q == 1)

VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>

Assembler Symbols

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

For <c> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

**Operation**

The description of [VORR (register)] gives the operational pseudocode for this instruction.
VRSHRN

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VRSHRN. The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | D | imm6 | Vd | 1 | 0 | 0 | 0 | 0 | 1 | M | 1 | Vm |

A1 (imm6 != 000xxx)

VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | D | imm6 | Vd | 1 | 0 | 0 | 0 | 0 | 1 | M | 1 | Vm |

T1 (imm6 != 000xxx)

VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> Is the data size for the elements of the vectors, encoded in "imm6<5:3>":

<table>
<thead>
<tr>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>16</td>
</tr>
<tr>
<td>01x</td>
<td>32</td>
</tr>
<tr>
<td>1xx</td>
<td>64</td>
</tr>
</tbody>
</table>
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

**Operation**

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount-1);
    for e = 0 to elements-1
        result = LSR(Elem[Qin][m>>1],e,2*esize) + round_const, shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector.

This is a pseudo-instruction of VMOVN. This means:

- The encodings in this description are named to match the encodings of VMOVN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VMOVN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operand, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>I16</td>
</tr>
<tr>
<td>01</td>
<td>I32</td>
</tr>
<tr>
<td>10</td>
<td>I64</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VMOVN gives the operational pseudocode for this instruction.
VRSQRTE

Vector Reciprocal Square Root Estimate finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

The operand and result elements are the same type, and can be floating-point numbers or unsigned integers. For details of the operation performed by this instruction see Floating-point reciprocal estimate and step.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VRSQRTE(<c>{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRSQRTE(<c>{<q>}.<dt> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
floating_point = (F == '1');

case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VRSQRTE(<c>{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRSQRTE(<c>{<q>}.<dt> <Qd>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');

case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler Symbols**

<
c>
For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q>
See *Standard assembler syntax fields*.

<dt>
Is the data type for the elements of the vectors, encoded in “F:size”:

<table>
<thead>
<tr>
<th>F size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 10</td>
<td>U32</td>
</tr>
<tr>
<td>1 01</td>
<td>F16</td>
</tr>
<tr>
<td>1 10</td>
<td>F32</td>
</tr>
</tbody>
</table>

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Newton-Raphson iteration**

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see *Floating-point reciprocal estimate and step*.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPRSqrtEstimate(Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = UnsignedRSqrtEstimate(Elem[D[m+r],e,esize]);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VRSQRTS

Vector Reciprocal Square Root Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the elements of the destination vector.

The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see Floating-point reciprocal estimate and step.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VRSQRTS{<c>{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VRSQRTS{<c>{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 0 1 1 1 1 0 D 1 sz Vn | Vd | 1 1 1 1 N Q M 1 Vm |

64-bit SIMD vector (Q == 0)

VRSQRTS{<c>{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

128-bit SIMD vector (Q == 1)

VRSQRTS{<c>{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' & & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & & !HaveFP16Ext() then UNDEFINED;

case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 0 1 1 1 1 0 D 1 sz Vn | Vd | 1 1 1 1 N Q M 1 Vm |

64-bit SIMD vector (Q == 0)

VRSQRTS{<c>{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

128-bit SIMD vector (Q == 1)

VRSQRTS{<c>{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' & & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & & !HaveFP16Ext() then UNDEFINED;
if sz == '1' & & InITBlock() then UNPREDICTABLE;

case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 0 1 1 1 1 0 D 1 sz Vn | Vd | 1 1 1 1 N Q M 1 Vm |

CONstrained UNPREDICTABLE behavior

If size == '01' & & InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration
For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see *Floating-point reciprocal estimate and step*.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRSqrtStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VRSRA

Vector Rounding Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the rounded results into the destination vector. For truncated results, see VSRA.

The operand and result elements must all be the same type, and can be any one of:
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 1   | 1   | 1   | 0   | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | U   |
| 1   | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | D   |
| imm6|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Vd  |
| 0   | 0   | 1   | 1   | L   | Q   | M   | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Vm  |

64-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 0)

VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>

128-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 1)

VRSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 1   | 1   | 1   | 1   | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | D   |
| imm6|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Vd  |
| 0   | 0   | 1   | 1   | L   | Q   | M   | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Vm  |

64-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 0)

VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>

128-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 1)

VRSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.
### Assembler Symbols

**<c>**
For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

**<q>**
See *Standard assembler syntax fields*.

**<type>**
Is the data type for the elements of the vectors, encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>&lt;type&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
</tr>
</tbody>
</table>

**<size>**
Is the data size for the elements of the vectors, encoded in "L:imm6<5:3>":

<table>
<thead>
<tr>
<th>L</th>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 01</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>0 1x</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>1 xx</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>xxx</td>
<td>64</td>
</tr>
</tbody>
</table>

**<Qd>**
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

**<Qm>**
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**<Dd>**
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**<Dm>**
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**<imm>**
Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

### Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D][m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D][d+r],e,esize] = Elem[D][d+r],e,esize] + result;
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Vector Rounding Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are rounded. For truncated results, see **VSUBHN**.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 D != 11 Vn</td>
</tr>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

<
  For encoding A1: see *Standard assemble syntax fields*. This encoding must be unconditional.
  For encoding T1: see *Standard assemble syntax fields*.

<<
  See *Standard assemble syntax fields*.

<dt>
  Is the data type for the elements of the operands, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>116</td>
</tr>
<tr>
<td>01</td>
<td>132</td>
</tr>
<tr>
<td>10</td>
<td>164</td>
</tr>
</tbody>
</table>

<Dd>
  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn>
  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Dot Product vector form with signed integers. 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.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

`ID ISAR6.DP` indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**
(Armv8.2)

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

64-bit SIMD vector (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

if !HaveDOTPExt() then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;

**T1**
(Armv8.2)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>Vn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>U</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
Assembler Symbols

See *Standard assembler syntax fields*.

- **<q>** Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- **<Qn>** Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- **<Qm>** Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- **<Dn>** Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- **<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```c
bits(64) operand1;
bits(64) operand2;
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            end if
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VSDOT (by element)

Dot Product index form with signed integers. 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.

In Armv8.2 and Armv8.3, this is an **OPTIONAL** instruction. From Armv8.4 it is mandatory for all implementations to support it.

**ID ISAR6.DP** indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1
(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  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | D | 1 | 0 | Vn | Vd | 1 | 1 | 0 | 1 | N | Q | M | 0 | Vm |

#### 64-bit SIMD vector (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

#### 128-bit SIMD vector (Q == 1)

VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]

if !HaveDOTPExt() then UNDEFINED;
if Q == '1' & Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;

### T1
(Armv8.2)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | D | 1 | 0 | Vn | Vd | 1 | 1 | 0 | 1 | N | Q | M | 0 | Vm |

#### 64-bit SIMD vector (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

#### 128-bit SIMD vector (Q == 1)

VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]

if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' & Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
Assembler Symbols

<q> See Standard assembler syntax fields.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

Operation

bits(64) operand1;
bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
**VSELEQ, VSELGE, VSELGT, VSELVS**

Floating-point conditional select allows the destination register to take the value in either one or the other source register according to the condition codes in the APSR.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | D  | cc | Vn | Vd | 1  | 0  | !=00| N  | 0  | M  | 0  | Vm |
```

size
VSELEQ, doubleprec (cc == 00 && size == 11)

VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

VSELEQ, halfprec (cc == 00 && size == 01)
(Armv8.2)

VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

VSELEQ, singleprec (cc == 00 && size == 10)

VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

VSELGE, doubleprec (cc == 10 && size == 11)

VSELGE.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

VSELGE, halfprec (cc == 10 && size == 01)
(Armv8.2)

VSELGE.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

VSELGE, singleprec (cc == 10 && size == 10)

VSELGE.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

VSELGT, doubleprec (cc == 11 && size == 11)

VSELGT.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

VSELGT, halfprec (cc == 11 && size == 01)
(Armv8.2)

VSELGT.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

VSELGT, singleprec (cc == 11 && size == 10)

VSELGT.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

VSELVS, doubleprec (cc == 01 && size == 11)

VSELVS.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

VSELVS, halfprec (cc == 01 && size == 01)
(Armv8.2)

VSELVS.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

VSELVS, singleprec (cc == 01 && size == 10)

VSELVS.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;

when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
cond = cc:(cc<1> EOR cc<0>):'0';
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>cc</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>!=</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VSELEQ, VSELGE, VSELGT, VSELVS
VSELEQ, doubleprec (cc == 00 && size == 11)
VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

VSELEQ, halfprec (cc == 00 && size == 01)
(Armv8.2)
VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

VSELEQ, singleprec (cc == 00 && size == 10)
VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

VSELGE, doubleprec (cc == 10 && size == 11)
VSELGE.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

VSELGE, halfprec (cc == 10 && size == 01)
(Armv8.2)
VSELGE.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

VSELGE, singleprec (cc == 10 && size == 10)
VSELGE.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

VSELGT, doubleprec (cc == 11 && size == 11)
VSELGT.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

VSELGT, halfprec (cc == 11 && size == 01)
(Armv8.2)
VSELGT.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

VSELGT, singleprec (cc == 11 && size == 10)
VSELGT.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

VSELVS, doubleprec (cc == 01 && size == 11)
VSELVS.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

VSELVS, halfprec (cc == 01 && size == 01)
(Armv8.2)
VSELVS.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

VSELVS, singleprec (cc == 01 && size == 10)
VSELVS.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
cond = cc:(cc<1> EOR cc<0>):'0';
CONSTRAINED UNPREDICTABLE behavior

If `INItBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Dd&gt;</td>
<td>Is the 64-bit name of the SIMD&amp;FP destination register, encoded in the &quot;D:Vd&quot; field.</td>
</tr>
<tr>
<td>&lt;Dn&gt;</td>
<td>Is the 64-bit name of the first SIMD&amp;FP source register, encoded in the &quot;N:Vn&quot; field.</td>
</tr>
<tr>
<td>&lt;Dm&gt;</td>
<td>Is the 64-bit name of the second SIMD&amp;FP source register, encoded in the &quot;M:Vm&quot; field.</td>
</tr>
<tr>
<td>&lt;Sd&gt;</td>
<td>Is the 32-bit name of the SIMD&amp;FP destination register, encoded in the &quot;Vd:D&quot; field.</td>
</tr>
<tr>
<td>&lt;Sn&gt;</td>
<td>Is the 32-bit name of the first SIMD&amp;FP source register, encoded in the &quot;Vn:N&quot; field.</td>
</tr>
<tr>
<td>&lt;Sm&gt;</td>
<td>Is the 32-bit name of the second SIMD&amp;FP source register, encoded in the &quot;Vm:M&quot; field.</td>
</tr>
</tbody>
</table>

Operation

```plaintext
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : (if ConditionHolds(cond) then S[n] else S[m])<15:0>;
    when 32
        S[d] = if ConditionHolds(cond) then S[n] else S[m];
    when 64
        D[d] = if ConditionHolds(cond) then D[n] else D[m];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector. Bits shifted out of the left of each element are lost. The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | D | imm6 | Vd | 0 | 1 | 0 | 1 | L | Q | M | 1 | Vm |

64-bit SIMD vector (!((imm6 == 000xxx && L == 0) && Q == 0))

VSHL{<c>}{<q>}.I{<size>}{<Dd>}, {<Dm>}, #<imm>

128-bit SIMD vector (!((imm6 == 000xxx && L == 0) && Q == 1))

VSHL{<c>}{<q>}.I{<size>}{<Qd>}, {<Qm>}, #<imm>

if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' & & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

| 1 | 1 | 1 | 1 | 1 | D | imm6 | Vd | 0 | 1 | 0 | 1 | L | Q | M | 1 | Vm |

### T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | D | imm6 | Vd | 0 | 1 | 0 | 1 | L | Q | M | 1 | Vm |

64-bit SIMD vector (!((imm6 == 000xxx && L == 0) && Q == 0))

VSHL{<c>}{<q>}.I{<size>}{<Dd>}, {<Dm>}, #<imm>

128-bit SIMD vector (!((imm6 == 000xxx && L == 0) && Q == 1))

VSHL{<c>}{<q>}.I{<size>}{<Qd>}, {<Qm>}, #<imm>

if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' & & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | D | imm6 | Vd | 0 | 1 | 0 | 1 | L | Q | M | 1 | Vm |

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.
Assembler Symbols

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

See Standard assembler syntax fields.

Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

<table>
<thead>
<tr>
<th>L imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 001</td>
<td>8</td>
</tr>
<tr>
<td>0 01x</td>
<td>16</td>
</tr>
<tr>
<td>0 1xx</td>
<td>32</td>
</tr>
<tr>
<td>1 xxx</td>
<td>64</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Is an immediate value, in the range 0 to <size>-1, encoded in the "imm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = LSL(Elem[D[m+r],e,esize], shift_amount);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

- The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VSHL (register)

Vector Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

For a rounding shift, see VRSHL.

The first operand and result elements are the same data type, and can be any one of:
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;

unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>U</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>

128-bit SIMD vector (Q == 1)

VSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;

unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in “U:size”: 
<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>0 11</td>
<td>S64</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
<tr>
<td>1 11</td>
<td>U64</td>
</tr>
</tbody>
</table>

<Qd> is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            result = Int(Elem[D[m+r],e,esize], unsigned) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Shift Left Long takes each element in a doubleword vector, left shifts them by an immediate value, and places the results in a quadword vector.

The operand elements can be:
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit untyped integers, maximum shift only.

The result elements are twice the length of the operand elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

$$\text{VSHLL}\{<c>\}<c>\{<q>\}.<\text{type}><\text{size}><Qd>,<Dm>,<\#imm>$$

if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = $\text{UInt}(\text{imm6}) - 8$;
  when '01xxxx' esize = 16; elements = 4; shift_amount = $\text{UInt}(\text{imm6}) - 16$;
  when '1xxxxx' esize = 32; elements = 2; shift_amount = $\text{UInt}(\text{imm6}) - 32$;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1'); d = $\text{UInt}(D:Vd)$; m = $\text{UInt}(M:Vm)$;

A2

$$\text{VSHLL}\{<c>\}<c>\{<q>\}.<\text{type}><\text{size}><Qd>,<Dm>,<\#imm>$$

if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << $\text{UInt}$(size); elements = $64 \div \text{esize}$; shift_amount = esize;
unsigned = FALSE; // Or TRUE without change of functionality
d = $\text{UInt}(D:Vd)$; m = $\text{UInt}(M:Vm)$;

T1

$$\text{VSHLL}\{<c>\}<c>\{<q>\}.<\text{type}><\text{size}><Qd>,<Dm>,<\#imm>$$

if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << $\text{UInt}$(size); elements = $64 \div \text{esize}$; shift_amount = esize;
unsigned = FALSE; // Or TRUE without change of functionality
d = $\text{UInt}(D:Vd)$; m = $\text{UInt}(M:Vm)$;
T1 (imm6 != 000xxx)

VSHLL\{<c}\{<q}\}.\langle\text{type}\rangle\langle\text{size}\rangle \langle\text{Qd}\rangle, \langle\text{Dm}\rangle, \#\langle\text{imm}\rangle

if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
  when '001xxx' \quad esize = 8; \quad elements = 8; \quad shift_amount = \text{UInt}(imm6) - 8;
  when '01xxxx' \quad esize = 16; \quad elements = 4; \quad shift_amount = \text{UInt}(imm6) - 16;
  when '1xxxxx' \quad esize = 32; \quad elements = 2; \quad shift_amount = \text{UInt}(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1'); \quad d = \text{UInt}(D:Vd); \quad m = \text{UInt}(M:Vm);

T2

VSHLL\{<c}\{<q}\}.\langle\text{type}\rangle\langle\text{size}\rangle \langle\text{Qd}\rangle, \langle\text{Dm}\rangle, \#\langle\text{imm}\rangle

if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 \quad elements = 64 \text{ DIV } esize; \quad shift_amount = esize;
unsigned = FALSE; \quad // Or TRUE without change of functionality
\quad d = \text{UInt}(D:Vd); \quad m = \text{UInt}(M:Vm);

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

\langle\text{c}\rangle \quad \text{For encoding A1 and A2: see Standard assembler syntax fields. This encoding must be unconditional.}
\text{For encoding T1 and T2: see Standard assembler syntax fields.}

\langle\text{q}\rangle \quad \text{See Standard assembler syntax fields.}

\langle\text{type}\rangle \quad \text{The data type for the elements of the operand. It must be one of:}
\quad \text{S} \quad \text{Signed. In encoding T1/A1, encoded as U = 0.}
\quad \text{U} \quad \text{Unsigned. In encoding T1/A1, encoded as U = 1.}
\quad \text{I} \quad \text{Untyped integer, Available only in encoding T2/A2.}

\langle\text{size}\rangle \quad \text{The data size for the elements of the operand. The following table shows the permitted values and their encodings:}
\begin{tabular}{|c|c|}
\hline
\langle\text{size}\rangle & \text{Encoding T1/A1} & \text{Encoding T2/A2} \\
\hline
8 & Encoded as imm6<5:3> = 0b001 & Encoded as size = 0b00 \\
16 & Encoded as imm6<5:4> = 0b01 & Encoded as size = 0b01 \\
32 & Encoded as imm6<5> = 1 & Encoded as size = 0b10 \\
\hline
\end{tabular}

\langle\text{Qd}\rangle \quad \text{Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \langle\text{Qd}\rangle*2.}

\langle\text{Dm}\rangle \quad \text{Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.}

\langle\text{imm}\rangle \quad \text{The immediate value. \langle\text{imm}\rangle must lie in the range 1 to \langle\text{size}\rangle, and:}
\quad \text{• If \langle\text{size}\rangle == \langle\text{imm}\rangle, the encoding is T2/A2.}
\quad \text{• Otherwise, the encoding is T1/A1, and:}
\hspace{1em} \text{• If \langle\text{size}\rangle == 8, \langle\text{imm}\rangle is encoded in imm6<2:0>.}
\hspace{1em} \text{• If \langle\text{size}\rangle == 16, \langle\text{imm}\rangle is encoded in imm6<3:0>.}
\hspace{1em} \text{• If \langle\text{size}\rangle == 32, \langle\text{imm}\rangle is encoded in imm6<4:0>.}
Operation

if `ConditionPassed()` then
    EncodingSpecificOperations(); `CheckAdvSIMDEnabled()`;
    for e = 0 to elements-1
        result = Int(Elem[Din][m],e,esize], unsigned) << shift_amount;
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Shift Right takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHR.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | U | 1 | D | imm6 | | Vd | 0 | 0 | 0 | 0 | L | Q | M | 1 | Vm |

**64-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 0)**

VSHR{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>

**128-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 1)**

VSHR{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | imm6 | | Vd | 0 | 0 | 0 | 0 | L | Q | M | 1 | Vm |

**64-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 0)**

VSHR{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>

**128-bit SIMD vector (!!(imm6 == 000xxx && L == 0) && Q == 1)**

VSHR{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.
Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> Is the data type for the elements of the vectors, encoded in “U”:

<table>
<thead>
<tr>
<th>U</th>
<th>&lt;type&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
</tr>
</tbody>
</table>

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

<table>
<thead>
<tr>
<th>L</th>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>1xx</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>xxx</td>
<td>64</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Shift Right copies the contents of one SIMD register to another.

This is a pseudo-instruction of VORR (register). This means:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

64-bit SIMD vector (Q == 0)

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{<dt>} <Dd>, <Dm>, <Dm>

128-bit SIMD vector (Q == 1)

VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{<dt>} <Qd>, <Qm>, <Qm>

T1

64-bit SIMD vector (Q == 0)

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{<dt>} <Dd>, <Dm>, <Dm>

128-bit SIMD vector (Q == 1)

VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{<dt>} <Qd>, <Qm>, <Qm>

Assembler Symbols

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.
<dt> Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

**Operation**

The description of [VORR (register)](#) gives the operational pseudocode for this instruction.

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VSHRN

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHRN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

A1 (imm6 != 000xxx)

VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);

T1

T1 (imm6 != 000xxx)

VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> Is the data size for the elements of the vectors, encoded in "imm6<5:3>"

<table>
<thead>
<tr>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>16</td>
</tr>
<tr>
<td>01x</td>
<td>32</td>
</tr>
<tr>
<td>1xx</td>
<td>64</td>
</tr>
</tbody>
</table>
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

**Operation**

if `ConditionPassed()` then

    EncodingSpecificOperations();  \( \text{CheckAdvSIMDEnabled}() \);

for e = 0 to elements-1

    result = \( \text{LSR}\left(\text{Elem}[Qin]\left(m\gg1\right),e,2*\text{esize}\right),\text{shift\_amount}\);

    \( \text{Elem}[D,d,e,\text{esize}] = \text{result}<\text{esize}-1:0>\);
**VSHRN (zero)**

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector.

This is a pseudo-instruction of **VMOVN**. This means:

- The encodings in this description are named to match the encodings of **VMOVN**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **VMOVN** gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
1  1  1  1  0  0  1  1  1 | D 1 1 | size 1 0 | Vd 0 0 1 0 0 | M 0 | Vm
```

### T1

```
1  1  1  1  1  1  1  1  1 | D 1 1 | size 1 0 | Vd 0 0 1 0 0 | M 0 | Vm
```

### Assembler Symbols

- `<c>` For encoding A1: see **Standard assembler syntax fields**. This encoding must be unconditional. For encoding T1: see **Standard assembler syntax fields**.
- `<q>` See **Standard assembler syntax fields**.
- `<dt>` Is the data type for the elements of the operand, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>I16</td>
</tr>
<tr>
<td>01</td>
<td>I32</td>
</tr>
<tr>
<td>10</td>
<td>I64</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.
- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as `<Qm>*2`.

### Operation

The description of **VMOVN** gives the operational pseudocode for this instruction.
Vector Shift Left and Insert takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost. The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

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

64-bit SIMD vector !(imm6 == 000xxx && L == 0) && Q == 0)

```
VSLI{c}<{q}.<size> {<Dd>},} <Dm>, #<imm>
```

128-bit SIMD vector !(imm6 == 000xxx && L == 0) && Q == 1)

```
VSLI{c}<{q}.<size> {<Dd>},} <Dm>, #<imm>
```

if (L:imm6) == '0001xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
  when '0001xxx'  
    esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
  when '001xxxxx'  
    esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
  when '01xxxxx'  
    esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
  when '1xxxxxx'  
    esize = 64;  elements = 1;  shift_amount = UInt(imm6);
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

### T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1 D</td>
</tr>
</tbody>
</table>

64-bit SIMD vector !(imm6 == 000xxx && L == 0) && Q == 0)

```
VSLI{c}<{q}.<size> {<Dd>},} <Dm>, #<imm>
```

128-bit SIMD vector !(imm6 == 000xxx && L == 0) && Q == 1)

```
VSLI{c}<{q}.<size> {<Dd>},} <Dm>, #<imm>
```

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
  when '0001xxx'  
    esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
  when '001xxxxx'  
    esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
  when '01xxxxx'  
    esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
  when '1xxxxxx'  
    esize = 64;  elements = 1;  shift_amount = UInt(imm6);
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.
Assembler Symbols

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

See Standard assembler syntax fields.

Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

<table>
<thead>
<tr>
<th>L</th>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>1xx</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>xxx</td>
<td>64</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    mask = LSL(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSL(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of signed 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an **OPTIONAL** instruction. **ID ISAR6**.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

(Armv8.6)

```
31 30 29 28 27 26 25 24 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 1 1 0 0 0 | D | 1 0 | Vn | Vd | 1 1 0 0 | N | 1 | M | 0 | Vm
```

<table>
<thead>
<tr>
<th>B</th>
<th>U</th>
</tr>
</thead>
</table>

### A1

VSMMLA\(<q>\).S8 \(<Qd>, \(<Qn>, \(<Qm>\)

if ![HaveAArch32Int8MatMulExt() then UNDEFINED;]

```asm
if ![HaveAArch32Int8MatMulExt() then UNDEFINED;]
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE; op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE; op2_unsigned = FALSE;
  when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Nv);
integer m = UInt(M:Vm);
```

### T1

(Armv8.6)

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 0 0 0 | D | 1 0 | Vn | Vd | 1 1 0 0 | N | 1 | M | 0 | Vm
```

<table>
<thead>
<tr>
<th>B</th>
<th>U</th>
</tr>
</thead>
</table>

### T1

VSMMLA\(<q>\).S8 \(<Qd>, \(<Qn>, \(<Qm>\)

if ![InitBlock() then UNPREDICTABLE;]

```asm
if ![HaveAArch32Int8MatMulExt() then UNDEFINED;]
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE; op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE; op2_unsigned = FALSE;
  when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Nv);
integer m = UInt(M:Vm);
```

### Assembler Symbols

\(<q>\) See *Standard assembler syntax fields*.  
\(<Qd>\) Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as \(<Qd>\)*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation**

```c
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend = Q[d>>1];
Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VSQRT

Square Root calculates the square root of the value in a floating-point register and writes the result to another floating-point register.

Depending on settings in the \texttt{CPACR}, \texttt{NSACR}, \texttt{HCPTR}, and \texttt{FPEXC} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see \emph{Enabling Advanced SIMD and floating-point support}.

It has encodings from the following instruction sets: A32 (\texttt{A1}) and T32 (\texttt{T1}).

\textbf{A1}

\begin{verbatim}
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 1 1 1 0 1 D 1 1 0 0 0 1</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{Half-precision scalar (size == 01)}
(Armv8.2)

\texttt{VSQRT\{<c}\{<q\}.F16 <Sd>, <Sm>}

\textbf{Single-precision scalar (size == 10)}

\texttt{VSQRT\{<c}\{<q\}.F32 <Sd>, <Sm>}

\textbf{Double-precision scalar (size == 11)}

\texttt{VSQRT\{<c}\{<q\}.F64 <Sd>, <Sm>}

\begin{verbatim}
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
case size of
  when '01' esize = 16; d = \texttt{UInt}(Vd:D); m = \texttt{UInt}(Vm:M);
  when '10' esize = 32; d = \texttt{UInt}(Vd:D); m = \texttt{UInt}(Vm:M);
  when '11' esize = 64; d = \texttt{UInt}(D:Vd); m = \texttt{UInt}(M:Vm);
\end{verbatim}

\textbf{CONSTRAINED UNPREDICTABLE behavior}

If \texttt{size == '01' && cond != '1110'}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

\textbf{T1}

\begin{verbatim}
<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 1 0 1 D 1 1 0 0 0 1</td>
</tr>
</tbody>
</table>
\end{verbatim}
Half-precision scalar (size == 01)
(Armv8.2)

VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONstrained UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

See Standard assembler syntax fields.

Operation

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  case esize of
    when 16 $d = Zeros(16) : FPSqrt($m<15:0>, FPSCR[]);
    when 32 $d = FPSqrt($m, FPSCR[]);
    when 64 $d = FPSqrt($m, FPSCR[]);
Vector Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the truncated results into the destination vector. For rounded results, see VRSRA.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

### A1

```
1 1 1 1 0 0 1 | U | D | imm6 | Vd | 0 0 0 1 | L | Q | M | 1 | Vm
```

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSRA{<c>}{<q>}.<type><size> {<Dd>},} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRA{<c>}{<q>}.<type><size> {<Qd>},} <Qm>, #<imm>
```

```c
if (L:imm6) == '0000xxx' then SEE "Related encodings";

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);

  unsigned = (U == '1');
  d = UInt(D:Vd);
  m = UInt(M:Vm);
  regs = if Q == '0' then 1 else 2;
```

### T1

```
1 1 1 U | 1 1 1 1 1 | D | imm6 | Vd | 0 0 0 1 | L | Q | M | 1 | Vm
```

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSRA{<c>}{<q>}.<type><size> {<Dd>},} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRA{<c>}{<q>}.<type><size> {<Qd>},} <Qm>, #<imm>
```

```c
if (L:imm6) == '0000xxx' then SEE "Related encodings";

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);

  unsigned = (U == '1');
  d = UInt(D:Vd);
  m = UInt(M:Vm);
  regs = if Q == '0' then 1 else 2;
```

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.
### Assembler Symbols

**<c>**
For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

**<q>**
See *Standard assembler syntax fields*.

**<type>**
Is the data type for the elements of the vectors, encoded in "U":

<table>
<thead>
<tr>
<th>U</th>
<th>&lt;type&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
</tr>
</tbody>
</table>

**<size>**
Is the data size for the elements of the vectors, encoded in "L:imm6<5:3>":

<table>
<thead>
<tr>
<th>L</th>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>1xx</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>xxx</td>
<td>64</td>
</tr>
</tbody>
</table>

**<Qd>**
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

**<Qm>**
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**<Dd>**
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**<Dm>**
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**<imm>**
Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

### Operation

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
      Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```

### Operational information

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

---

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VSRI

Vector Shift Right and Insert takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 1 1 D imm6 Vd 0 1 0 0 L Q M 1 Vm

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

VSRI{<c>{<q>}.<size> {<Dd>,} <Dm>, #<imm>

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

VSRI{<c>{<q>}.<size> {<Qd>,} <Qm>, #<imm>

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

case L:imm6 of
  when '0001xxx'  esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx'  esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx'  esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx'  esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
  
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 D imm6 Vd 0 1 0 0 L Q M 1 Vm

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

VSRI{<c>{<q>}.<size> {<Dd>,} <Dm>, #<imm>

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

VSRI{<c>{<q>}.<size> {<Qd>,} <Qm>, #<imm>

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

case L:imm6 of
  when '0001xxx'  esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx'  esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx'  esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx'  esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
  
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Related encodings: See Advanced SIMD one register and modified immediate for the T32 instruction set, or Advanced SIMD one register and modified immediate for the A32 instruction set.
Assembler Symbols

For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

See *Standard assembler syntax fields*.

Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

<table>
<thead>
<tr>
<th>L</th>
<th>imm6&lt;5:3&gt;</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>1xx</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>xxx</td>
<td>64</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

Operation

```pseudocode
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    mask = LSR(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSR(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
VST1 (single element from one lane)

Store single element from one lane of one register stores one element to memory from one element of a register. For details of the addressing mode see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2 and A3) and T32 (T1, T2 and T3).

A1

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

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); alignment = 1;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

A2

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

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
### Offset (Rm == 1111)

VST1<\{<c>\}<\{<q>\}.<size> <\list>, [<\{<Rn>\},:<\{<align>\}>]

### Post-indexed (Rm == 1101)

VST1<\{<c>\}<\{<q>\}.<size> <\list>, [<\{<Rn>\},:<\{<align>\}>]]

### Post-indexed (Rm != 11x1)

VST1<\{<c>\}<\{<q>\}.<size> <\list>, [<\{<Rn>\},:<\{<align>\}>], <\{<Rm>\>

if size == '11' then UNDEFINED;
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' & index_align<1:0> != '11' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>); alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 then UNPREDICTABLE;

### T1

| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | D | 0 | 0 | Rn | Vd | 1 | 0 | 0 | 0 | index_align | Rm |

### Offset (Rm == 1111)

VST1<\{<c>\}<\{<q>\}.<size> <\list>, [<\{<Rn>\},:<\{<align>\}>]

### Post-indexed (Rm == 1101)

VST1<\{<c>\}<\{<q>\}.<size> <\list>, [<\{<Rn>\},:<\{<align>\}>]]

### Post-indexed (Rm != 11x1)

VST1<\{<c>\}<\{<q>\}.<size> <\list>, [<\{<Rn>\},:<\{<align>\}>], <\{<Rm>\>

if size == '11' then UNDEFINED;
if index_align<2> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); alignment = 1;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 then UNPREDICTABLE;

### T2

| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | D | 0 | 0 | Rn | Vd | 0 | 1 | 0 | 0 | index_align | Rm |
Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

T3

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then UNDEFINED;
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> For encoding A1, A2 and A3: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1, T2 and T3: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> Is the data size, encoded in “size”:

VST1 (single element from one lane)
Is a list containing the single 64-bit name of the SIMD&FP register holding the element. The list must be \{ \text{<Dd>\[<index>\]} \}. The register <Dd> is encoded in the "D:Vd" field. The permitted values and encoding of <index> depend on <size>:

\begin{itemize}
\item \textbf{<size> == 8} \hfill \text{<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.}
\item \textbf{<size> == 16} \hfill \text{<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.}
\item \textbf{<size> == 32} \hfill \text{<index> is 0 or 1, encoded in the "index_align<3>" field.}
\end{itemize}

<\textbf{Rn}> Is the general-purpose base register, encoded in the "Rn" field.

<\textbf{align}> When \textbf{<size> == 8}, <\text{align}> must be omitted, otherwise it is the optional alignment. Whenever \text{<align>} is omitted, the standard alignment is used, see \textit{Unaligned data access}, and the encoding depends on \textbf{<size>}:

\begin{itemize}
\item \textbf{<size> == 8} \hfill \text{Encoded in the "index_align<0>" field as 0.}
\item \textbf{<size> == 16} \hfill \text{Encoded in the "index_align<1:0>" field as 0b00.}
\item \textbf{<size> == 32} \hfill \text{Encoded in the "index_align<2:0>" field as 0b000.}
\end{itemize}

Whenever \text{<align>} is present, the permitted values and encoding depend on \textbf{<size>}:

\begin{itemize}
\item \textbf{<size> == 16} \hfill \text{<align> is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as 0b01.}
\item \textbf{<size> == 32} \hfill \text{<align> is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as 0b011.}
\end{itemize}

: is the preferred separator before the <\text{align}> value, but the alignment can be specified as @\text{<align>}, see \textit{Advanced SIMD addressing mode}.

<\textbf{Rm}> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see \textit{Advanced SIMD addressing mode}.

\textbf{Operation}

\begin{verbatim}
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address,ebytes] = Elem[R[d],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
    
Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VST1 (multiple single elements)

Store multiple single elements from one, two, three, or four registers stores elements to memory from one, two, three, or four registers, without interleaving. Every element of each register is stored. For details of the addressing mode see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2, A3 and A4) and T32 (T1, T2, T3 and T4).

A1

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

Offset (Rm == 1111)

```
VST1{<c>}<{q}>.<size> <list>, [<Rn>{{<align>}}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}<{q}>.size <list>, [<Rn>{{<align>}}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}<{q}>.size <list>, [<Rn>{{<align>}}], <Rm>
```

CONRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 1 0 0 0 D 0 0 | Rn | Vd | 1 0 1 0 | size | align | Rm
```
Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>
```

- regs = 2; if align == '11' then UNDEFINED;
- alignment = if align == '00' then 1 else 4 << UInt(align);
- ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
- d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
- wback = (m != 15); register_index = (m != 15 & m != 13);
- if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

<table>
<thead>
<tr>
<th>D</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Vd</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>align</th>
<th>Rm</th>
</tr>
</thead>
</table>

Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>
```

- regs = 3; if align<1> == '1' then UNDEFINED;
- alignment = if align == '00' then 1 else 4 << UInt(align);
- ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
- d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
- wback = (m != 15); register_index = (m != 15 & m != 13);
- if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

regs = 1; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:
• The instruction is **UNDEFINED**.
• The instruction executes as **NOP**.
• The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

### T2

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Offset (Rm == 1111)**

VSTI{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}

**Post-indexed (Rm == 1101)**

VSTI{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

**Post-indexed (Rm != 11x1)**

VSTI{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
regs = 2; if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

• The instruction is **UNDEFINED**.
• The instruction executes as **NOP**.
• The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

### T3

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vd</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VSTI (multiple single elements)
Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

  regs = 3;  if align < 1 == '1' then UNDEFINED;
  alignment = if align == '00' then 1 else 4 << UINT(align);
  ebytes = 1 << UINT(size);  elements = 8 DIV ebytes;
  d = UINT(D:Vd);  n = UINT(Rn);  m = UINT(Rm);
  wback = (m != 15);  register_index = (m != 15 & m != 13);

if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T4

```
  15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 2 1 0
  0 0 0 1 0 1 0 0 1 D 0 0 Rn Vd 0 0 1 0 size align Rm
```

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

  regs = 4;
  alignment = if align == '00' then 1 else 4 << UINT(align);
  ebytes = 1 << UINT(size);  elements = 8 DIV ebytes;
  d = UINT(D:Vd);  n = UINT(Rn);  m = UINT(Rm);
  wback = (m != 15);  register_index = (m != 15 & m != 13);

if n == 15 || d+regs > 32 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST1 (multiple single elements).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

<c> For encoding A1, A2, A3 and A4: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1, T2, T3 and T4: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> Is the data size, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>64</td>
</tr>
</tbody>
</table>

<list> Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:

{ <Dd> }  
Single register. Selects the A1 and T1 encodings of the instruction.

{ <Dd>, <Dd+1> }  
Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.

{ <Dd>, <Dd+1>, <Dd+2> }  
Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }  
Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as 0b00.
Whenever <align> is present, the permitted values are:

64  64-bit alignment, encoded in the "align" field as 0b01.

128 128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.

256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about <Rn>, !, and <Rm>, see Advanced SIMD addressing mode.
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if ebytes != 8 then
                MemU[address,ebytes] = Elem[D[d+r],e];
            else
                - = AArch32.CheckAlignment(address, ebytes, AccType_NORMAL, iswrite);
                MemU[address,4] = if BigEndian(AccType_NORMAL) then data<63:32> else data<31:0>;
                MemU[address+4,4] = if BigEndian(AccType_NORMAL) then data<31:0> else data<63:32>;
                address = address + ebytes;
            if wback then
                if register_index then
                    R[n] = R[n] + R[m];
                else
                    R[n] = R[n] + 8*regs;
        if wback then
            if register_index then
                R[n] = R[n] + R[m];
            else
                R[n] = R[n] + 8*regs;
VST2 (single 2-element structure from one lane)

Store single 2-element structure from one lane of two registers stores one 2-element structure to memory from corresponding elements of two registers. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2 and A3) and T32 (T1, T2 and T3).

A1

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

Offset (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then UNDEFINED;
   ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
   alignment = if index_align<0> == '0' then 1 else 2;
   d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
   wback = (m != 15); register_index = (m != 15 && m != 13);
   if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDITABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------------------|-------|-------|-----------------|
| 1 1 1 1 0 1 0 0 1 | D | 0 | 0 | Rn | Vd | 0 | 0 | 0 | 1 | index_align | Rm |
```
Offset (Rm == 1111)

VST2{{c}\{q}.<size> <list>, [<Rn>{:<align}]}}

Post-indexed (Rm == 1101)

VST2{{c}\{q}.<size> <list>, [<Rn>{:<align}]}}!

Post-indexed (Rm != 11x1)

VST2{{c}\{q}.<size> <list>, [<Rn>{:<align}]}, <Rm>

if size == '11' then UNDEFINED;
    ebytes = 2;  index = UInt(index_align<3:2>);
    inc = if index_align<1> == '0' then 1 else 2;
    alignment = if index_align<0> == '0' then 1 else 4;
    d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
    wback = (m != 15);  register_index = (m != 15 && m != 13);
    if n == 15 || d2 > 31 then UNPREDICTABLE;

CONstrained UNpredictable behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior
does not affect any other memory locations.

A3

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

size

Offset (Rm == 1111)

VST2{{c}\{q}.<size> <list>, [<Rn>{:<align}]}}

Post-indexed (Rm == 1101)

VST2{{c}\{q}.<size> <list>, [<Rn>{:<align}]}}!

Post-indexed (Rm != 11x1)

VST2{{c}\{q}.<size> <list>, [<Rn>{:<align}]}, <Rm>

if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
    ebytes = 4;  index = UInt(index_align<3>);
    inc = if index_align<2> == '0' then 1 else 2;
    alignment = if index_align<0> == '0' then 1 else 8;
    d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
    wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONstrained UNpredictable behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

**T1**

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Offset (Rm == 1111)**

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

**Post-indexed (Rm == 1101)**

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

**Post-indexed (Rm != 11x1)**

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

**CONstrained UNPREDICTABLE behavior**

If d2 > 31, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

**T2**

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Offset (Rm == 1111)

VST2{<c}{{q}.<size> <list>, [<Rn>{{:align}}

Post-indexed (Rm == 1101)

VST2{<c}{{q}.<size> <list>, [<Rn>{{:align}}

Post-indexed (Rm != 1111)

VST2{<c}{{q}.<size> <list>, [<Rn>{{:align}}}, <Rm>

if size == '11' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 0 0 1 1 D 0 0 | Rn | Vd | 1 0 0 1 | index_align | Rm |
|-----------------------------------------|------------------------|---|----|----------|-------------|
| size                                    |                        |   |    |          |             |

Offset (Rm == 1111)

VST2{<c}{{q}.<size> <list>, [<Rn>{{:align}}

Post-indexed (Rm == 1101)

VST2{<c}{{q}.<size> <list>, [<Rn>{{:align}}

Post-indexed (Rm != 1111)

VST2{<c}{{q}.<size> <list>, [<Rn>{{:align}}}, <Rm>

if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

VST2 (single 2-element structure from one lane)
The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly **VST2 (single 2-element structure from one lane)**.

**Assembler Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;c&gt;</code></td>
<td>For encoding A1, A2 and A3: see <em>Standard assembler syntax fields</em>. This encoding must be unconditional. For encoding T1, T2 and T3: see <em>Standard assembler syntax fields</em>.</td>
</tr>
<tr>
<td><code>&lt;q&gt;</code></td>
<td>See <em>Standard assembler syntax fields</em>.</td>
</tr>
</tbody>
</table>
| `<size>` | Is the data size, encoded in "size":
| size | `<size>` |
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| `<list>` | Is a list containing the 64-bit names of the two SIMD&FP registers holding the element. The list must be one of:
| List | Description |
| { `<Dd>[<index>], <Dd+1>[<index>]` } | Single-spaced registers, encoded as "spacing" = 0. |
| { `<Dd>[<index>], <Dd+2>[<index>]` } | Double-spaced registers, encoded as "spacing" = 1. Not permitted when `<size>` == 8. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| `<align>` | Is the optional alignment. Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access*, and the encoding depends on `<size>`:
| `<size>` | Description |
| 00 | Encoded in the "index_align<0>" field as 0. |
| 01 | Encoded in the "index_align<0>" field as 0. |
| 10 | Encoded in the "index_align<1:0>" field as 0b00. |
| 11 | `<align>` is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1. |
| 12 | `<align>` is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1. |
<size> == 32
<align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address, ebytes] = Elem[D[d], index];
    MemU[address+ebytes, ebytes] = Elem[D[d2], index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
    else
        R[n] = R[n] + 2*ebytes;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VST2 (multiple 2-element structures)

Store multiple 2-element structures from two or four registers stores multiple 2-element structures from two or four registers to memory, with interleaving. For more information, see Element and structure load/store instructions. Every element of each register is saved. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

Itype

Offset (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

reg = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(aligned);
bytes = 1 << UInt(size); elements = 8 DIV bytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d2+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  1 | 1  | 1  | 1  | 0  | 0  | 0  | 0  | D  | 0  | 0  | Rn | Vd | 0  | 0  | 1  | 1  | size | align | Rm |
Offset (Rm == 1111)

VST2\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST2\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 111)

VST2\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

CONSTRANDED UNPREDICTABLE behavior

If d2+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

If n == 15 || d2+regs > 32 then UNPREDICTABLE;

VST2 (multiple 2-element structures)
The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Offset (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

```
regs = 2; inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2+regs > 32 then UNPREDICTABLE;
```
Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1001.

Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

Is the general-purpose base register, encoded in the "Rn" field.

Is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as 0b00. Whenever <align> is present, the permitted values are:

64 64-bit alignment, encoded in the "align" field as 0b01.
128 128-bit alignment, encoded in the "align" field as 0b10.
256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            MemU[address, ebytes] = Elem[D[d+r], e];
            MemU[address+ebytes, ebytes] = Elem[D[d2+r], e];
            address = address + 2*ebytes;
        if wback then
            if register_index then
                R[n] = R[n] + R[m];
            else
                R[n] = R[n] + 16*regs;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VST3 (single 3-element structure from one lane)

Store single 3-element structure from one lane of three registers stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode see Advanced SIMD addressing mode. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1, A2 and A3) and T32 (T1, T2 and T3).

A1

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

Offset (Rm == 1111)

VST3{<c>{<q>.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>{<q>.<size> <list>, [<Rn>]

Post-indexed (Rm != 11x1)

VST3{<c>{<q>.<size> <list>, [<Rn>], <Rm

if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONstrained UnPREDICtable behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | D  | 0  | 0  | Rn | Vd | 0  | 1  | 1  | 0  | index_align | Rm | size |
Offset (Rm == 1111)

VST3{<c}>{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c}>{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

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

Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNDEFINED. If the instruction specifies writeback, then that register becomes UNDEFINED. This behavior does not affect any other memory locations.

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1</td>
</tr>
</tbody>
</table>

size

Offset (Rm == 1111)

VST3{<c>{<q}>.<size> <list>}, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>{<q}>.<size> <list>}, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>{<q}>.<size> <list>}, [<Rn>], <Rm>

if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRUINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNDEFINED. If the instruction specifies writeback, then that register becomes UNDEFINED. This behavior does not affect any other memory locations.

T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1</td>
</tr>
</tbody>
</table>

size
Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
  ebytes = 2;  index = UInt(index_align<3:2>);
  inc = if index_align<1> == '0' then 1 else 2;
  d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
  wback = (m != 15);  register_index = (m != 15 && m != 13);
  if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
  inc = if index_align<2> == '0' then 1 else 2;
  d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
  wback = (m != 15);  register_index = (m != 15 && m != 13);
  if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST3 (single 3-element structure from one lane).

Assembler Symbols

<c>  For encoding A1, A2 and A3: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1, T2 and T3: see Standard assembler syntax fields.

<q>  See Standard assembler syntax fields.

<size>  Is the data size, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
</tbody>
</table>

<list>  Is a list containing the 64-bit names of the three SIMD&FP registers holding the element. The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }  Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }  Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 8
"spacing" is encoded in the "index_align<0>" field.

<size> == 16
"spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.

<size> == 32
"spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn>  Is the general-purpose base register, encoded in the "Rn" field.

<Rm>  Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Alignment

Standard alignment rules apply, see Alignment support.
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n];
    MemU[address, ebytes] = Elem[D[d], index];
    MemU[address+ebytes, ebytes] = Elem[D[d2], index];
    MemU[address+2*ebytes, ebytes] = Elem[D[d3], index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VST3 (multiple 3-element structures)

Store multiple 3-element structures from three registers stores multiple 3-element structures to memory from three registers, with interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be *UNDEFINED*, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

**Offset (Rm == 1111)**

\[
\text{VST3}\{<c}\}{<q}\}.<\text{size}> <\text{list}>, [<\text{Rn}>{:<\text{align}>}]
\]

**Post-indexed (Rm == 1101)**

\[
\text{VST3}\{<c}\}{<q}\}.<\text{size}> <\text{list}>, [<\text{Rn}>{:<\text{align}>}]
\]

**Post-indexed (Rm != 11x1)**

\[
\text{VST3}\{<c}\}{<q}\}.<\text{size}> <\text{list}>, [<\text{Rn}>{:<\text{align}>}], <\text{Rm}>
\]

\[
\text{if size == '11' || align<1> == '1' then UNDEFINED; case itype of}
\]

\[
\text{when '0100'}
\]

\[
\text{inc = 1;}
\]

\[
\text{when '0101'}
\]

\[
\text{inc = 2;}
\]

\[
\text{otherwise} \quad \text{SEE "Related encodings";}
\]

\[
\text{alignment = if align<0> == '0' then 1 else 8;}
\]

\[
\text{ebbytes = 1 << \text{UInt(size); elements = 8 DIV ebytes;}}
\]

\[
\text{d = UInt(D:Vd); } \text{d2 = d + inc; } \text{d3 = d2 + inc; } \text{n = UInt(Rn); } \text{m = UInt(Rm);}
\]

\[
\text{wback = (m != 15); register_index = (m != 15 } \&\& m != 13);
\]

\[
\text{if n == 15 } || \text{ d3 > 31 then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If d3 > 31, then one of the following behaviors must occur:

- The instruction is *UNDEFINED*.
- The instruction executes as *NOP*.
- The memory locations specified by the instruction and the number of registers specified by the instruction become *UNKNOWN*. If the instruction specifies writeback, then that register becomes *UNKNOWN*. This behavior does not affect any other memory locations.

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | D  | 0  | 0  | Rn | Vd | 0  | 1  | 0  | x  | size | align | Rm |

**itype**
Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' || align<1> == '1' then UNDEFINED;
  case itype of
  when '0100'
    inc = 1;
  when '0101'
    inc = 2;
  otherwise
    SEE "Related encodings";
  alignment = if align<0> == '0' then 1 else 8;
  ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
  d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
  wback = (m != 15);  register_index = (m != 15 && m != 13);
  if n == 15 || d3 > 31 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If d3 > 31, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

For more information about the CONSTRAINTED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly VST3 (multiple 3-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

**Assembler Symbols**

- For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
- For encoding T1: see *Standard assembler syntax fields*.
- See *Standard assembler syntax fields*.
- Is the data size, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- Is a list containing the 64-bit names of the SIMD&FP registers.
  The list must be one of:
  
  { <Dd>, <Dd+1>, <Dd+2> }
  
  Single-spaced registers, encoded in the "itype" field as 0b0100.

  { <Dd>, <Dd+2>, <Dd+4> }
  
  Double-spaced registers, encoded in the "itype" field as 0b0101.

  The register <Dd> is encoded in the "D:Vd" field.

- Is the general-purpose base register, encoded in the "Rn" field.
Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access, and is encoded in the "align" field as 0b00.
Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as 0b01.
: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        MemU[address, ebytes] = Elem[D[d], e];
        MemU[address+ebytes, ebytes] = Elem[D[d2], e];
        MemU[address+2*ebytes, ebytes] = Elem[D[d3], e];
        address = address + 3*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**VST4 (single 4-element structure from one lane)**

Store single 4-element structure from one lane of four registers stores one 4-element structure to memory from corresponding elements of four registers. For details of the addressing mode see *Advanced SIMD addressing mode*. Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ( **A1**, **A2** and **A3**) and T32 ( **T1**, **T2** and **T3**).

**A1**

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

**Offset (Rm == 1111)**

\[
\text{VST4}\{\text{<c>}\}\{\text{<q>}\}.\text{<size>} \ \text{<list>}, \ [\text{<Rn>}{:\text{<align}}>]
\]

**Post-indexed (Rm == 1101)**

\[
\text{VST4}\{\text{<c>}\}\{\text{<q>}\}.\text{<size>} \ \text{<list>}, \ [\text{<Rn>}{:\text{<align}}>]!
\]

**Post-indexed (Rm != 11x1)**

\[
\text{VST4}\{\text{<c>}\}\{\text{<q>}\}.\text{<size>} \ \text{<list>}, \ [\text{<Rn>}{:\text{<align}}>], \ <Rm>
\]

\[
\text{if size == '11' then UNDEFINED;}
\text{if size != '00' then SEE "Related encodings";}
\text{ebytes = 1; index = UInt(index_align<3:1>); inc = 1;}
\text{alignment = if index_align<0> == '0' then 1 else 4;}
\text{d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);}
\text{wback = (m != 15); register_index = (m != 15 && m != 13);}
\text{if n == 15 || d4 > 31 then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(d4 > 31\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

**A2**

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

**VST4 (single 4-element structure from one lane)**
Offset (Rm == 1111)

\[
VST4\{<c>\{<q>\}.<size> <list>, [<Rn\{:<align}>]}
\]

Post-indexed (Rm == 1101)

\[
VST4\{<c>\{<q>\}.<size> <list>, [<Rn\{:<align}>]}
\]

Post-indexed (Rm != 11x1)

\[
VST4\{<c>\{<q>\}.<size> <list>, [<Rn\{:<align}>], <Rm
\]

if size == '11' then UNDEFINED;
if size != '01' then SEE "Related encodings";
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONstrained UNpredictable behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writes back, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | 1  | D  | 0  | 0  | Rn | Vd | 1  | 0  | 1  | 1  | index_align | Rm |

Offset (Rm == 1111)

\[
VST4\{<c>\{<q>\}.<size> <list>, [<Rn\{:<align}>]}
\]

Post-indexed (Rm == 1101)

\[
VST4\{<c>\{<q>\}.<size> <list>, [<Rn\{:<align}>]}
\]

Post-indexed (Rm != 11x1)

\[
VST4\{<c>\{<q>\}.<size> <list>, [<Rn\{:<align}>], <Rm
\]

if size == '11' then UNDEFINED;
if size != '01' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONstrained UNpredictable behavior

If d4 > 31, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then UNDEFINED;
if size != '00' then SEE "Related encodings";
ebytes = 1; index = U16(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = U16(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = U16(Rn); m = U16(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONstrained unPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

VST4 (single 4-element structure from one lane)
Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then UNDEFINED;
if size != '01' then SEE "Related encodings";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is_UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
</tr>
</tbody>
</table>

size

Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is_UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
• The instruction is **UNDEFINED**.
• The instruction executes as **NOP**.
• The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly **VST4 (single 4-element structure from one lane)**.

**Assembler Symbols**

<c>
For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1, T2 and T3: see *Standard assembler syntax fields*.

$q$
See *Standard assembler syntax fields*.

<size>
Is the data size, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
</tbody>
</table>

<list>
Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.
The list must be one of:

\{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] \}  
Single-spaced registers, encoded as “spacing” = 0.

\{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] \}  
Double-spaced registers, encoded as “spacing” = 1. Not permitted when <size> == 8.

The encoding of “spacing” depends on <size>:

\(<size> == 16\)  
“spacing” is encoded in the “index_align<1>” field.

\(<size> == 32\)  
“spacing” is encoded in the “index_align<2>” field.

The register <Dd> is encoded in the “D:Vd” field.
The permitted values and encoding of <index> depend on <size>:

\(<size> == 8\)  
<index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.

\(<size> == 16\)  
<index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.

\(<size> == 32\)  
<index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn>
Is the general-purpose base register, encoded in the “Rn” field.

<align>
Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and the encoding depends on <size>:

\(<size> == 8\)  
Encoded in the “index_align<0>” field as 0.

\(<size> == 16\)  
Encoded in the “index_align<0>” field as 0.

\(<size> == 32\)  
Encoded in the “index_align<1:0>” field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

\(<size> == 8\)  
<align> is 32, meaning 32-bit alignment, encoded in the “index_align<0>” field as 1.

\(<size> == 16\)  
<align> is 64, meaning 64-bit alignment, encoded in the “index_align<0>” field as 1.
<size> == 32

<align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see Advanced SIMD addressing mode.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see Advanced SIMD addressing mode.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    address = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address, ebytes] = Elem[D[d], index];
    MemU[address+ebytes, ebytes] = Elem[D[d2], index];
    MemU[address+2*ebytes, ebytes] = Elem[D[d3], index];
    MemU[address+3*ebytes, ebytes] = Elem[D[d4], index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
    end
end
VST4 (multiple 4-element structures)

Store multiple 4-element structures from four registers stores multiple 4-element structures to memory from four registers, with interleaving. For more information, see Element and structure load/store instructions. Every element of each register is saved. For details of the addressing mode see Advanced SIMD addressing mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

itype

Offset (Rm == 1111)

```
VST4{<c>}{<q>},{<size>},{<list>}, [{<Rn}>{:<<align>}>]}
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>},{<size>},{<list>}, [{<Rn}>{:<<align>}>]}
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>},{<size>},{<list>}, [{<Rn}>{:<<align>}>}, <Rm>
```

if size == '11' then UNDEFINED;
case itype of
  when '0000'
    inc = 1;
  when '0001'
    inc = 2;
  otherwise
    SEE "Related encodings";
alignment = if align == '00' then 1 else 4 <-> UInt(align);
ebytes = 1 <-> UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

```
   15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  | 1 1 1 1 0 1 0 0 | D 0 0 | Rn | Vd | 0 0 0 | x | size | align | Rm |
```

itype
Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

if size == '11' then UNDEFINED;
case itype of
  when '0000'
    inc = 1;
  when '0001'
    inc = 2;
  otherwise
    SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly VST4 (multiple 4-element structures).

Related encodings: See Advanced SIMD element or structure load/store for the T32 instruction set, or Advanced SIMD element or structure load/store for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<size> Is the data size, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<list> Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:

- \{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> \}
  Single-spaced registers, encoded in the "itype" field as 0b0000.

- \{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> \}
  Double-spaced registers, encoded in the "itype" field as 0b0001.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.
Is the optional alignment. Whenever \(<align>\) is omitted, the standard alignment is used, see \textit{Unaligned data access}, and is encoded in the "align" field as 0b00.

Whenever \(<align>\) is present, the permitted values are:

\begin{itemize}
  \item \textbf{64} \hfill 64-bit alignment, encoded in the "align" field as 0b01.
  \item \textbf{128} \hfill 128-bit alignment, encoded in the "align" field as 0b10.
  \item \textbf{256} \hfill 256-bit alignment, encoded in the "align" field as 0b11.
\end{itemize}

\(<align>\) is the preferred separator before the \(<align>\) value, but the alignment can be specified as \(@<align>\), see \textit{Advanced SIMD addressing mode}.

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see \textit{Advanced SIMD addressing mode}.

\textbf{Operation}

\begin{verbatim}
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  address = R[n];  iswrite = TRUE;
  address = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
  for e = 0 to elements-1
    MemU[address, ebytes] = Elem[D[d], e];
    MemU[address+ebytes, ebytes] = Elem[D[d2], e];
    MemU[address+2*ebytes, ebytes] = Elem[D[d3], e];
    MemU[address+3*ebytes, ebytes] = Elem[D[d4], e];
    address = address + 4*ebytes;
  if wback then
    if register_index then
      R[n] = R[n] + R[m];
    else
      R[n] = R[n] + 32;
\end{verbatim}

VSTM, VSTMDB, VSTMIA

Store multiple SIMD&FP registers stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register.
Depending on settings in the CPACR, NSACR, HCPTR, and FPACR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

This instruction is used by the alias VPUSH.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
!= 1111 1 1 0 | P | U | D | W | 0 | Rn | Vd | 1 0 1 1 | imm8<7:1> | 0
```

Decrement Before (P == 1 && U == 0 && W == 1)

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Increment After (P == 0 && U == 1)

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

if P == '0' && U == '0' && W == '1' then SEE "Related encodings";
if P == '1' && U == '0' && W == '0' then SEE "VSTR";
if P == U && U == W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_reg = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:00', 32);
regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 16 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If regs > 16 || (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
!= 1111 1 1 0 | P | U | D | W | 0 | Rn | Vd | 1 0 1 0 | imm8
```

Decrement Before (P == 1 & U == 0 & W == 1)

VSTMDB{<c}>{<q}{{<size>} <Rn>!, <sreglist>

Increment After (P == 0 & U == 1)

VSTM{<c}>{<q}{{<size>} <Rn>{!}, <sreglist>

VSTMIA{<c}>{<q}{{<size>} <Rn>{!}, <sreglist>

if P == '0' & U == '0' & W == '0' then SEE "Related encodings";
if P == '1' & W == '0' then SEE "VSTR";
if P == U & W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 & (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

<table>
<thead>
<tr>
<th>P</th>
<th>U</th>
<th>D</th>
<th>W</th>
<th>Rn</th>
<th>Vd</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>imm8&lt;7:1&gt;</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

Decrement Before (P == 1 & U == 0 & W == 1)

VSTMDB{<c}>{<q}{{<size>} <Rn>!, <dreglist>

Increment After (P == 0 & U == 1)

VSTM{<c}>{<q}{{<size>} <Rn>{!}, <dreglist>

VSTMIA{<c}>{<q}{{<size>} <Rn>{!}, <dreglist>

if P == '0' & U == '0' & W == '0' then SEE "Related encodings";
if P == '1' & W == '0' then SEE "VSTR";
if P == U & W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 & (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || imm8<0> == '1' || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If regs == 0, then one of the following behaviors must occur:
The instruction is UNDEFINED.
The instruction executes as NOP.
The instruction operates as a VSTM with the same addressing mode but stores no registers.

If \( \text{regs} > 16 \) \( \text{||} \) \( \text{(d+regs)} > 32 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

### T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>D</td>
<td>W</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Decrement Before \((P == 1 \&\& U == 0 \&\& W == 1)\)

VSTMDB\{<c>\}{<q>}{.<size>} \(<Rn>!\), <sreglist>

Increment After \((P == 0 \&\& U == 1)\)

VSTM\{<c>\}{<q>}{.<size>} \(<Rn>!\), <sreglist>

VSTMIA\{<c>\}{<q>}{.<size>} \(<Rn>!\), <sreglist>

if \(P == '0' \&\& U == '0' \&\& W == '0'\) then SEE "Related encodings";
if \(P == '1' \&\& U == '0' \&\& W == '0'\) then SEE "VSTR";
if \(P == U \&\& W == '1'\) then UNDEFINED;

// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE; add = \((U == '1')\); wback = \((W == '1')\); \(d = \text{UINT}(Vd:D)\); \(n = \text{UINT}(Rn)\);
imm32 = \text{ZeroExtend}(imm8:'00', 32); \(\text{regs} = \text{UINT}(imm8)\);
if \(n == 15 \&\& \text{wback} \text{|| CurrentInstrSet()} \neq \text{InstrSet_A32}\) then UNPREDICTABLE;
if \(\text{regs} == 0 \text{||} (d+\text{regs}) > 32\) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(\text{regs} == 0\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If \((d+\text{regs}) > 32\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly VSTM.

Related encodings: See *Advanced SIMD and floating-point 64-bit move* for the T32 instruction set, or *Advanced SIMD and floating-point 64-bit move* for the A32 instruction set.

**Assembler Symbols**

- \(<c>\) See *Standard assembler syntax fields*.
- \(<q>\) See *Standard assembler syntax fields*.
- \(<\text{size}>\) An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
<Rn> Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

! Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

<sreglist> Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

<dreglist> Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D-Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPUSH</td>
<td>P == '1' &amp;&amp; U == '0' &amp;&amp; W == '1' &amp;&amp; Rn == '1101'</td>
</tr>
</tbody>
</table>

### Operation

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r]; address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
        if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VSTR

Store SIMD&FP register stores a single register from the Advanced SIMD and floating-point register file to memory, using an address from a general-purpose register, with an optional offset. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
<table>
<thead>
<tr>
<th>!= 1111</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>U</th>
<th>D</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

cond

**Half-precision scalar (size == 01)**
(Armv8.2)

VSTR{<c>}{<q>}.16 <5d>, [<Rn>{, #/{+/-}<imm>}]

**Single-precision scalar (size == 10)**

VSTR{<c>}{<q>}{.32} <5d>, [<Rn>{, #/{+/-}<imm}>]

**Double-precision scalar (size == 11)**

VSTR{<c>}{<q>}{.64} <5d>, [<Rn>{, #/{+/-}<imm}>]

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
  n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T1**

```
<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>U</th>
<th>D</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Half-precision scalar (size == 01)
(Armv8.2)

VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #{+/-}<imm}>]

Single-precision scalar (size == 10)

VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm}>]

Double-precision scalar (size == 11)

VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm}>]

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
inImm = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;

CONSTRUED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRUED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<c> See Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

.64 Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Dd> Is the 64-bit name of the SIMD&FP source register, encoded in the "D:Vd" field.

.32 Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Sd> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vd:D" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

<table>
<thead>
<tr>
<th>U</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

<imm> For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the “imm8” field as <imm>/4.
For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the “imm8” field as <imm>/2.
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    case esize of
        when 16
            MemA[address,2] = $d<15:0>;
        when 32
            MemA[address,4] = $d;
        when 64
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian(AccType_ATOMIC) then $d<63:32> else $d<31:0>;
            MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then $d<31:0> else $d<63:32>;

VSUB (floating-point)

Vector Subtract (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector. Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1

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

64-bit SIMD vector (Q == 0)

VSUB{<c>}{<q>}.{dt} {<Dd>,} {<Dn>,} {<Dm>}

128-bit SIMD vector (Q == 1)

VSUB{<c>}{<q>}.{dt} {<Qd>,} {<Qn>,} {<Qm>}

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
    case sz of
        when '0' esize = 32; elements = 2;
        when '1' esize = 16; elements = 4;
    d = Uint(D:Vd);  n = Uint(N:Vn);  m = Uint(M:Vm);  regs = if Q == '0' then 1 else 2;

A2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| != 1111 | 1 | 1 | 1 | 0 | 0 | D | 1 | 1 | Vn | Vd | 1 | 0 | size | N | 1 | M | 0 | Vm |

cond

Half-precision scalar (size == 01)
(Armv8.2)

VSUB{<c>}{<q>}.F16 {<Sd>,} {<Sn>,} {<Sm>}

Single-precision scalar (size == 10)

VSUB{<c>}{<q>}.F32 {<Sd>,} {<Sn>,} {<Sm>}

Double-precision scalar (size == 11)

VSUB{<c>}{<q>}.F64 {<Dd>,} {<Dn>,} {<Dm>}

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
    case size of
        when '01' esize = 16; d = Uint(Vd:D); n = Uint(Vn:N);  m = Uint(Vm:M);
        when '10' esize = 32; d = Uint(Vd:D); n = Uint(Vn:N);  m = Uint(Vm:M);
        when '11' esize = 64; d = Uint(D:Vd);  n = Uint(N:Vn);  m = Uint(M:Vm);
CONSTRANGED UNPREDICTABLE behavior

If $\text{size} == \text{`01'} \&& \text{cond} != \text{`110'},$ then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector ($Q == 0$)

$\text{VSUB}{\{c}\{q}\}.\{dt\} \{\{Dd\}, \{Dn\}, \{Dm\}}$

128-bit SIMD vector ($Q == 1$)

$\text{VSUB}{\{c}\{q}\}.\{dt\} \{\{Qd\}, \{Qn\}, \{Qm\}}$

if $Q == \text{`1'} \&& (\text{Vd}<0> == \text{`1'} \| \| \text{Vn}<0> == \text{`1'} \| \| \text{Vm}<0> == \text{`1'})$ then **UNDEFINED**;
if $sz == \text{`1'} \&& \!\text{HaveFP16Ext()}$ then **UNDEFINED**;
if $sz == \text{`1'} \&& \!\text{InITBlock()}$ then **UNPREDICTABLE**;
advsimd = TRUE;

```c
case sz of
    when `0' esize = 32; elements = 2;
    when `1' esize = 16; elements = 4;
d = Uint(D:Vd); n = Uint(N:Vn); m = Uint(M:Vm); regs = if Q == `0' then 1 else 2;
```

CONSTRANGED UNPREDICTABLE behavior

If $sz == \text{`1'} \&& \!\text{InITBlock()}$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$\text{size}$ $\text{N}$ $\text{M}$ $\text{Vn}$ $\text{Vd}$ $\text{Vm}$

VSUB (floating-point)
Half-precision scalar (size == 01)
(Armv8.2)

VSUB{<c>}{<q>}.F16 {<Sd>,}  <Sn>, <Sm>

Single-precision scalar (size == 10)

VSUB{<c>}{<q>}.F32 {<Sd>,}  <Sn>, <Sm>

Double-precision scalar (size == 11)

VSUB{<c>}{<q>}.F64 {<Dd>,}  <Dn>, <Dm>

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONstrained UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding A2, T1 and T2: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F32</td>
</tr>
<tr>
<td>1</td>
<td>F16</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPSub(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRVal);
    else             // VFP instruction
        case esize of
            when 16
                S[d] = Zeros(16) : FPSub(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            when 32
                S[d] = FPSub(S[n], S[m], FPSCR[]);
            when 64
                D[d] = FPSub(D[n], D[m], FPSCR[]);
```


**VSUB (integer)**

Vector Subtract (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector. Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ( **A1** ) and T32 ( **T1** ).

### A1

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

**64-bit SIMD vector (Q == 0)**

\[
\text{VSUB}\{<c>\}{<q>}.<dt> \{<Dd>, }<Dn>, <Dm>
\]

**128-bit SIMD vector (Q == 1)**

\[
\text{VSUB}\{<c>\}{<q>}.<dt> \{<Qd>, }<Qn>, <Qm>
\]

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

\[
esize = 8 \ll \text{UInt}(\text{size}); \text{ elements } = \text{64 DIV esize};
\]

\[
d = \text{UInt}(D:Vd); \quad n = \text{UInt}(N:Vn); \quad m = \text{UInt}(M:Vm); \quad \text{regs } = \text{if } Q == '0' \text{ then 1 else 2};
\]

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | D  | size | Vn | Vd | 1  | 0  | 0  | 0  | N | Q | M | 0 | Vm |

**64-bit SIMD vector (Q == 0)**

\[
\text{VSUB}\{<c>\}{<q>}.<dt> \{<Dd>, }<Dn>, <Dm>
\]

**128-bit SIMD vector (Q == 1)**

\[
\text{VSUB}\{<c>\}{<q>}.<dt> \{<Qd>, }<Qn>, <Qm>
\]

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

\[
esize = 8 \ll \text{UInt}(\text{size}); \text{ elements } = \text{64 DIV esize};
\]

\[
d = \text{UInt}(D:Vd); \quad n = \text{UInt}(N:Vn); \quad m = \text{UInt}(M:Vm); \quad \text{regs } = \text{if } Q == '0' \text{ then 1 else 2};
\]

### Assembler Symbols

- `<c>`: For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
- `<q>`: See *Standard assembler syntax fields*.
- `<dt>`: Is the data type for the elements of the vectors, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>I8</td>
</tr>
<tr>
<td>01</td>
<td>I16</td>
</tr>
<tr>
<td>10</td>
<td>I32</td>
</tr>
<tr>
<td>11</td>
<td>I64</td>
</tr>
</tbody>
</table>

- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D-Vd" field as `<Qd>*2.`
- `<Qn>`: Is the 128-bit name of the first SIMD&FP source register, encoded in the "N-Vn" field as `<Qn>*2.`
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] - Elem[D[m+r],e,esize];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

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

Vector Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are truncated. For rounded results, see VRSUBHN.

There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 0 1 D != 11 Vn Vd 0 1 1 0 N 0 M 0 Vm
size
```

A1

```
VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

if size == '11' then SEE “Related encodings”;
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

```
 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 1 1 1 1 D != 11 Vn Vd 0 1 1 0 N 0 M 0 Vm
size
```

T1

```
VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

if size == '11' then SEE “Related encodings”;
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

```
<c>
For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

<q>
See Standard assembler syntax fields.

<dt>
Is the data type for the elements of the operands, encoded in “size”:

```
<dt> size
00 116
01 132
10 164
```

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>+2.

<Qm>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>+2.
if **ConditionPassed**() then
   EncodingSpecificOperations();  **CheckAdvSIMDEnabled**();
   for e = 0 to elements-1
      result = **Elem**[Qin[n>>1],e,2*esize] - **Elem**[Qin[m>>1],e,2*esize];
      **Elem**[D[d],e,esize] = result<2*esize-1:esize>;

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Subtract Long subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in a quadword vector. Before subtracting, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

size op
```

**T1**

```
<table>
<thead>
<tr>
<th>15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
</tr>
</tbody>
</table>

size op
```

**Assembler Symbols**

For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

For the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

### Related encodings

- **Encoding A1:**
  - If size == '11' then SEE “Related encodings”;
  - If Vd<0> == '1' | (op == '1' && Vn<0> == '1') then UNDEFINED;
  - unsigned = (U == '1');
  - esize = 8 << UInt(size); elements = 64 DIV esize; is_vsubw = (op == '1');
  - d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

- **Encoding T1:**
  - If size == '11' then SEE “Related encodings”;
  - If Vd<0> == '1' | (op == '1' && Vn<0> == '1') then UNDEFINED;
  - unsigned = (U == '1');
  - esize = 8 << UInt(size); elements = 64 DIV esize; is_vsubw = (op == '1');
  - d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

### Related encodings:
For Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

<table>
<thead>
<tr>
<th>U size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
</tr>
</tbody>
</table>
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for e = 0 to elements-1
    if is_vsubw then
      op1 = Int(Elem[Qin][n>>1],e,2*esize], unsigned);
    else
      op1 = Int(Elem[Din][n],e,esize], unsigned);
    result = op1 - Int(Elem[Din][m],e,esize], unsigned);
    Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see **Enabling Advanced SIMD and floating-point support**.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

### A1

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
1 1 1 1 0 0 1 |U| 1 |D| != 11 |Vn| Vd | 0 0 1 | 1 |N| 0 |M| 0 | Vm |
```

```
size op
```

### T1

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
1 1 1 |U| 1 1 1 1 1 1 |D| != 11 |Vn| Vd | 0 0 1 | 1 |N| 0 |M| 0 | Vm |
```

```
size op
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

### Assembler Symbols

- **<c>** For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  
- For encoding T1: see *Standard assembler syntax fields*.
- **<q>** See *Standard assembler syntax fields*.
- **<dt>** Is the data type for the elements of the second operand vector, encoded in “U:size”:

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>S8</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>S16</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>S32</td>
<td></td>
</tr>
<tr>
<td>1 00</td>
<td>U8</td>
<td></td>
</tr>
<tr>
<td>1 01</td>
<td>U16</td>
<td></td>
</tr>
<tr>
<td>1 10</td>
<td>U32</td>
<td></td>
</tr>
</tbody>
</table>
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vsubw then
            op1 = Int(Elem[Qin][n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din][n],e,esize], unsigned);
        result = op1 - Int(Elem[Din][m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Dot Product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1
(Armv8.6)

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

64-bit SIMD vector (Q == 0)

VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector (Q == 1)

VSUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>[<index>]

if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;

T1
(Armv8.6)

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 1 D 0 0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector (Q == 0)

VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector (Q == 1)

VSUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>[<index>]

if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
Assembler Symbols

See *Standard assembler syntax fields*.

- **<qd>** Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- **<qn>** Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- **<Dn>** Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- **<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
- **<index>** Is the element index in the range 0 to 1, encoded in the "M" field.

Operation

```c
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = Int(Elem[operand1, 4 * e + b, 8], op1_unsigned);
            element2 = Int(Elem[operand2, 4 * i + b, 8], op2_unsigned);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Vector Swap exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

64-bit SIMD vector (Q == 0)

VSWP<c>{<q})){.<dt}> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VSWP<c>{<q})){.<dt}> <Qd>, <Qm>

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

```
15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0 15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
1  1  1  1  1  1  1  1  1  D  1  1  0  0  1  1  Vd  0  0  0  0  Q  M  0  Vm
```

64-bit SIMD vector (Q == 0)

VSWP<c>{<q})){.<dt}> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VSWP<c>{<q})){.<dt}> <Qd>, <Qm>

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**Assembler Symbols**

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

&q> See Standard assembler syntax fields.

<dt> An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            D[d+r] = Din[m+r];
            D[m+r] = Din[d+r];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
VTBL, VTBX

Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.
Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.
Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

VTBL (op == 0)

VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>

VTBX (op == 1)

VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>

is_vtbl = (op == '0'); length = UInt(len)+1;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If n + length > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | D  | 1  | 1  | Vn | Vd | 1  | 0  | len | N  | op | M  | 0  | Vm |

VTBL (op == 0)

VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>

VTBX (op == 1)

VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>

is_vtbl = (op == '0'); length = UInt(len)+1;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If n + length > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
The instruction executes as NOP.
One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

- `<c>` For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
  For encoding T1: see Standard assembler syntax fields.
- `<q>` See Standard assembler syntax fields.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.
- `<list>` The vectors containing the table. It must be one of:
  - `{<Dn>}`
    Encoded as len = 0b00.
  - `{<Dn>, <Dn+1>}`
    Encoded as len = 0b01.
  - `{<Dn>, <Dn+1>, <Dn+2>}`
    Encoded as len = 0b10.
  - `{<Dn>, <Dn+1>, <Dn+2>, <Dn+3>}`
    Encoded as len = 0b11.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register holding the indices, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();

  // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
  table3 = if length == 4 then D[n+3] else Zeros(64);
  table2 = if length >= 3 then D[n+2] else Zeros(64);
  table1 = if length >= 2 then D[n+1] else Zeros(64);
  table = table3 : table2 : table1 : D[n];

  for i = 0 to 7
    index = UInt(Elem[D[m],i,8]);
    if index < 8*length then
      Elem[D[d],i,8] = Elem[table,index,8];
    else
      if is_vtbl then
        Elem[D[d],i,8] = Zeros(8);
      // else Elem[D[d],i,8] unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Transpose treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows an example of the operation of VTRN doubleword operations.

<table>
<thead>
<tr>
<th>VTRN.32</th>
<th>VTRN.16</th>
<th>VTRN.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dd</td>
<td>Dd</td>
<td>Dd</td>
</tr>
<tr>
<td>m</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>1 0</td>
<td>3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

Dn   Dn   Dn

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instructions **VUZP (alias)**, and **VZIP (alias)**.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

```
0 1 1 1 1 0 0 1 1 1 D 1 1 size 1 0 Vd 0 0 0 1 Q M 0 Vm
```

64-bit SIMD vector (Q == 0)

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

```
0 1 1 1 1 1 1 1 1 1 1 D 1 1 size 1 0 Vd 0 0 0 1 Q M 0 Vm
```

64-bit SIMD vector (Q == 0)

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
Assembler Symbols

<_c>_ For encoding A1: see _Standard assembler syntax fields_. This encoding must be unconditional.
For encoding T1: see _Standard assembler syntax fields_.
<_q>_ See _Standard assembler syntax fields_.
<_dt>_ Is the data type for the elements of the vectors, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<_Qd>_ Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<_Qm>_ Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<_Dd>_ Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<_Dm>_ Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations(); _CheckAdvSIMDEnabled_();
    h = elements DIV 2;
    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            for e = 0 to h-1
                Elem[D[d+r],2*e+1,esize] = Elem[Din[m+r],2*e,esize];
                Elem[D[m+r],2*e,esize] = Elem[Din[d+r],2*e+1,esize];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VTST

Vector Test Bits takes each element in a vector, and bitwise ANDs it with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:
- 8-bit, 16-bit, or 32-bit fields.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | D  | size | Vn | Vd | 1  | 0  | 0  | 0  | N  | Q  | M  | 1  | Vm |

64-bit SIMD vector (Q == 0)

VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields.

;q> See Standard assembler syntax fields.

<dt> Is the data type for the elements of the operands, encoded in “size”:
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
</tbody>
</table>

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Db> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !IsZero(Elem[D[n+r],e,esize] AND Elem[D[m+r],e,esize]) then
                Elem[D[d+r],e,esize] = Ones(esize);
            else
                Elem[D[d+r],e,esize] = Zeros(esize);
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Dot Product vector form with unsigned integers. 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.

In Armv8.2 and Armv8.3, this is an **OPTIONAL** instruction. From Armv8.4 it is mandatory for all implementations to support it.

**ID ISAR6.DP** indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

(Armv8.2)

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

**64-bit SIMD vector (Q == 0)**

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>

if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;

**T1**

(Armv8.2)

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 0 0 D 1 0</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector (Q == 0)**

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>

**128-bit SIMD vector (Q == 1)**

VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
Assembler Symbols

See *Standard assembler syntax fields*.

- `<Q>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- `<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.
- `<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.
- `<D>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```c
bits(64) operand1;
bits(64) operand2;
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
  operand1 = D[n+r];
  operand2 = D[m+r];
  result = D[d+r];
  integer element1, element2;
  for e = 0 to 1
    integer res = 0;
    for i = 0 to 3
      if signed then
        element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      else
        element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
  D[d+r] = result;
```

Internal version only: isa v01.19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VUDOT (by element)

Dot Product index form with unsigned integers. 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.

In Armv8.2 and Armv8.3, this is an \textit{OPTIONAL} instruction. From Armv8.4 it is mandatory for all implementations to support it.

\textbf{ID ISAR6.DP} indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

\textbf{A1 (Armv8.2)}

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 0 | 0 | D | 1 | 0 | Vn | Vd | 1 1 0 | 1 | N | Q | M | 1 | Vm
\end{verbatim}

\textbf{64-bit SIMD vector (Q == 0)}

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

\textbf{128-bit SIMD vector (Q == 1)}

VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

\begin{verbatim}
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
\end{verbatim}

\textbf{T1 (Armv8.2)}

\begin{verbatim}
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 0 | 0 | D | 1 | 0 | Vn | Vd | 1 1 0 | 1 | N | Q | M | 1 | Vm
\end{verbatim}

\textbf{64-bit SIMD vector (Q == 0)}

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

\textbf{128-bit SIMD vector (Q == 1)}

VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

\begin{verbatim}
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
\end{verbatim}
Assembler Symbols

<q> See Standard assembler syntax fields.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

Operation

```cpp
bits(64) operand1;
bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
  operand1 = D[n+r];
  result = D[d+r];
  integer element1, element2;
  for e = 0 to 1
    integer res = 0;
    for i = 0 to 3
      if signed then
        element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
      else
        element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
      res = res + element1 * element2;
      Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```
VUMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an **OPTIONAL** instruction. **ID ISAR6**.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1
(Armv8.6)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
B 1 1 1 1 1 1 1 0 0 0 | D 1 0 | Vn | Vd | 1 1 0 0 | N 1 | M 1 | Vm
```

A1

```
VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE; op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE; op2_unsigned = FALSE;
  when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

T1
(Armv8.6)

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
B 1 1 1 1 1 1 1 0 0 0 | D 1 0 | Vn | Vd | 1 1 0 0 | N 1 | M 1 | Vm
```

T1

```
VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE; op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE; op2_unsigned = FALSE;
  when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

**Assembler Symbols**

- `<q>` See [Standard assembler syntax fields](#).
- `<Qd>` Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as `<Qd>*2.`
Operation

```c
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend = Q[d>>1];
Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
VUSDOT (vector)

Dot Product vector form with mixed-sign integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an **OPTIONAL** instruction. **ID_ISAR6**.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**
(Armv8.6)

```
1 1 1 1 1 1 1 0 0 1 D 1 0 Vn Vd 1 1 0 1 N Q M 0 Vm
```

64-bit SIMD vector (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

```ceml
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

**T1**
(Armv8.6)

```
31 30 29 28 27 26 25 24 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 1 1 0 0 1 D 1 0 Vn Vd 1 1 0 1 N Q M 0 Vm
```

64-bit SIMD vector (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

```ceml
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

**Assembler Symbols**

- `<q>` See *Standard assembler syntax fields*.
- `<Qd>` Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as `<Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the first SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```c
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;
for r = 0 to regs-1
    operand1 = Din[n+r];
    operand2 = Din[m+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = UInt(Elem[operand1, 4 * e + b, 8]);
            element2 = SInt(Elem[operand2, 4 * e + b, 8]);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VUSDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an **OPTIONAL** instruction. **ID ISAR6.I8MM** indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

**A1**
(Armv8.6)

| 31 30 29 28 27 26 25 24 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 | 1 | 1 | 1 | 0 | 1 | D | 0 | 0 | Vn | Vd | 1 | 1 | 0 | 1 | N | Q | M | 0 | Vm |
| U |

64-bit SIMD vector (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector (Q == 1)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

if ![HaveAArch32Int8MatMulExt() then UNDEFINED;](if ![HaveAArch32Int8MatMulExt() then UNDEFINED;]

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;

boolean op1_unsigned = (U == '0');

boolean op2_unsigned = (U == '1');

integer d = UInt(D:Vd);

integer n = UInt(N:Vn);

integer m = UInt(Vm);

integer i = UInt(M);

integer regs = if Q == '1' then 2 else 1;

**T1**
(Armv8.6)

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | D | 0 | 0 | Vn | Vd | 1 | 1 | 0 | 1 | N | Q | M | 0 | Vm |
| U |

64-bit SIMD vector (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector (Q == 1)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

if ![InITBlock() then UNPREDICTABLE;](if ![InITBlock() then UNPREDICTABLE;]

if ![HaveAArch32Int8MatMulExt() then UNDEFINED;](if ![HaveAArch32Int8MatMulExt() then UNDEFINED;]

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;

boolean op1_unsigned = (U == '0');

boolean op2_unsigned = (U == '1');

integer d = UInt(D:Vd);

integer n = UInt(N:Vn);

integer m = UInt(Vm);

integer i = UInt(M);

integer regs = if Q == '1' then 2 else 1;
**Assembler Symbols**

- **<q>** See *Standard assembler syntax fields*.
- **<Qd>** Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- **<Qn>** Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.
- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- **<Dn>** Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- **<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
- **<index>** Is the element index in the range 0 to 1, encoded in the "M" field.

**Operation**

```plaintext
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;
operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = Int(Elem[operand1, 4 * e + b, 8], op1_unsigned);
            element2 = Int(Elem[operand2, 4 * i + b, 8], op2_unsigned);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3 ; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an \texttt{OPTIONAL} instruction. \texttt{ID ISAR6.I8MM} indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (\texttt{A1}) and T32 (\texttt{T1}).

\textbf{A1} (Armv8.6)

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 0 0 1 D 1 0 Vn Vd 1 1 0 0 N 1 M 0 Vm
B U
\end{verbatim}

\textbf{T1} (Armv8.6)

\begin{verbatim}
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 0 0 1 D 1 0 Vn Vd 1 1 0 0 N 1 M 0 Vm
B U
\end{verbatim}

\textbf{Assembler Symbols}

\begin{itemize}
  \item \texttt{<q>}: See \textit{Standard assembler syntax fields}.
  \item \texttt{<Qd>}: Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as \texttt{<Qd>\*2}.
\end{itemize}
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation**

```c
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend    = Q[d>>1];
Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Internal version only: isa v01_19, pseudocode v2020-09_xml, sve v2020-09_rc3; Build timestamp: 2020-09-30T21:35

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
VUZP

Vector Unzip de-interleaves the elements of two vectors. The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types. The following figure shows an example of the operation of VUZP doubleword operation for data type 8.

VUZP, doubleword

<table>
<thead>
<tr>
<th>Register state before operation</th>
<th>Register state after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dd A7 A6 A5 A4 A3 A2 A1 A0 B6 B4</td>
<td>Dd A7 A6 A5 A4 A3 A2 A1 A0 B6 B4</td>
</tr>
<tr>
<td>Dm B7 B6 B5 B4 B3 B2 B1 B0</td>
<td>Dm B7 B6 B5 B4 B3 B2 B1 B0</td>
</tr>
<tr>
<td>A7 A5 A3 A1</td>
<td>A7 A5 A3 A1</td>
</tr>
</tbody>
</table>

The following figure shows an example of the operation of VUZP quadword operation for data type 32.

VUZP, quadword

<table>
<thead>
<tr>
<th>Register state before operation</th>
<th>Register state after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qd A3 A2 A1 A0 B2 B0 A2 A0</td>
<td>Qd A3 A2 A1 A0 B2 B0 A2 A0</td>
</tr>
<tr>
<td>Qm B3 B2 B1 B0 B3 B1 A3 A1</td>
<td>Qm B3 B2 B1 B0 B3 B1 A3 A1</td>
</tr>
</tbody>
</table>

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

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

64-bit SIMD vector (Q == 0)

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadrword_operation = (Q == '1'); esize = 8 << UInt(size);
d = UInt(D:Vd); m = UInt(M:Vm);

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------------------------|------------------------------------------|
| 1 1 1 1 1 1 1 1 1 | D 1 1 | size 1 0 | Vd 0 0 0 1 0 | Q M 0 | Vm |

64-bit SIMD vector (Q == 0)

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadrword_operation = (Q == '1'); esize = 8 << UInt(size);
d = UInt(D:Vd); m = UInt(M:Vm);

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>1x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN; Q[m>>1] = bits(128) UNKNOWN;
        else
            zipped_q = Q[m>>1]:Q[d>>1];
            for e = 0 to (128 DIV esize) - 1
                Elem[Q[d>>1],e,esize] = Elem[zipped_q,2*e,esize];
                Elem[Q[m>>1],e,esize] = Elem[zipped_q,2*e+1,esize];
        else
            if d == m then
                D[d] = bits(64) UNKNOWN; D[m] = bits(64) UNKNOWN;
            else
                zipped_d = D[m]:D[d];
                for e = 0 to (64 DIV esize) - 1
                    Elem[D[d],e,esize] = Elem[zipped_d,2*e,esize];
                    Elem[D[m],e,esize] = Elem[zipped_d,2*e+1,esize];
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Unzip de-interleaves the elements of two vectors.

This is a pseudo-instruction of VTRN. This means:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

**A1**

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

64-bit SIMD vector

\[ \text{VUZP}\{<c>\}{<q>}.32 \quad \text{<Dd>, \ <Dm>} \]

is equivalent to

\[ \text{VTRN}\{<c>\}{<q>}.32 \quad \text{<Dd>, \ <Dm>} \]

**T1**

```
| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | size | 1 | 0 | Vd | 0 | 0 | 0 | 0 | 1 | 0 | M | 0 | Vm |
```

64-bit SIMD vector

\[ \text{VUZP}\{<c>\}{<q>}.32 \quad \text{<Dd>, \ <Dm>} \]

is equivalent to

\[ \text{VTRN}\{<c>\}{<q>}.32 \quad \text{<Dd>, \ <Dm>} \]

**Assembler Symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
  For encoding T1: see *Standard assembler syntax fields*.
- `<q>` See *Standard assembler syntax fields*.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation**

The description of VTRN gives the operational pseudocode for this instruction.
Vector Zip interleaves the elements of two vectors. The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types. The following figure shows an example of the operation of VZIP doubleword operation for data type 8.

VZIP.8, doubleword

<table>
<thead>
<tr>
<th>Register state before operation</th>
<th>Register state after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD</td>
<td>A7</td>
</tr>
<tr>
<td>Dm</td>
<td>B7</td>
</tr>
</tbody>
</table>

The following figure shows an example of the operation of VZIP quadword operation for data type 32.

VZIP.32, quadword

<table>
<thead>
<tr>
<th>Register state before operation</th>
<th>Register state after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qd</td>
<td>A3</td>
</tr>
<tr>
<td>Qm</td>
<td>B3</td>
</tr>
</tbody>
</table>

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

64-bit SIMD vector (Q == 0)

VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>

T1

64-bit SIMD vector (Q == 0)

VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

See *Standard assembler syntax fields*.

For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>1x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;dt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "M:Vm" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "D:Vd" field.

**Operation**

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  if quadword_operation then
    if d == m then
      Q[d>>1] = bits(128) UNKNOWN; Q[m>>1] = bits(128) UNKNOWN;
    else
      bits(256) zipped_q;
      for e = 0 to (128 DIV esize) - 1
        Elem[zipped_q,2*e,esize] = Elem[Q[d>>1],e,esize];
        Elem[zipped_q,2*e+1,esize] = Elem[Q[m>>1],e,esize];
      Q[d>>1] = zipped_q<127:0>; Q[m>>1] = zipped_q<255:128>;
    else
      if d == m then
        D[d] = bits(64) UNKNOWN; D[m] = bits(64) UNKNOWN;
      else
        bits(128) zipped d;
        for e = 0 to (64 DIV esize) - 1
          Elem[zipped d,2*e,esize] = Elem[D[d],e,esize];
          Elem[zipped d,2*e+1,esize] = Elem[D[m],e,esize];
        D[d] = zipped_d<63:0>; D[m] = zipped_d<127:64>;
  else
    if d == m then
      D[d] = bits(64) UNKNOWN; D[m] = bits(64) UNKNOWN;
    else
      bits(128) zipped d;
      for e = 0 to (64 DIV esize) - 1
        Elem[zipped d,2*e,esize] = Elem[D[d],e,esize];
        Elem[zipped d,2*e+1,esize] = Elem[D[m],e,esize];
      D[d] = zipped_d<63:0>; D[m] = zipped_d<127:64>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

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

Vector Zip interleaves the elements of two vectors.

This is a pseudo-instruction of VTRN. This means:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

|   | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | D | 1 | 1 | size | 1 | 0 | Vd | 0 | 0 | 0 | 0 | 1 | 0 | M | 0 | Vm |
| Q |

64-bit SIMD vector

VZIP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.32 <Dd>, <Dm>

T1

|   | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | size | 1 | 0 | Vd | 0 | 0 | 0 | 0 | 1 | 0 | M | 0 | Vm |
| Q |

64-bit SIMD vector

VZIP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.32 <Dd>, <Dm>

Assembler Symbols

<c> For encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields.

<q> See Standard assembler syntax fields.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

The description of VTRN gives the operational pseudocode for this instruction.
Top-level encodings for A32

<table>
<thead>
<tr>
<th>cond</th>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 00x</td>
<td>Data-processing and miscellaneous instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!= 1111 010</td>
<td>Load/Store Word, Unsigned Byte (immediate, literal)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!= 1111 011 0</td>
<td>Load/Store Word, Unsigned Byte (register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!= 1111 011 1</td>
<td>Media instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10x</td>
<td>Branch, branch with link, and block data transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11x</td>
<td>System register access, Advanced SIMD, floating-point, and Supervisor call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111 0xx</td>
<td>Unconditional instructions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data-processing and miscellaneous instructions

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>cond</th>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 00</td>
<td>Extra load/store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0xxxx</td>
<td>Multiply and Accumulate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1xxxx</td>
<td>Synchronization primitives and Load-Acquire/Store-Release</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 10xx0</td>
<td>Miscellaneous</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 10xx0 1</td>
<td>Halfword Multiply and Accumulate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 != 10xx0 0</td>
<td>Data-processing register (immediate shift)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 != 10xx0 0 1</td>
<td>Data-processing register (register shift)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Data-processing immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extra load/store

These instructions are under Data-processing and miscellaneous instructions.

<table>
<thead>
<tr>
<th>cond</th>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 000</td>
<td>Load/Store Dual, Half, Signed Byte (register)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Load/Store Dual, Half, Signed Byte (register, literal)</td>
<td></td>
</tr>
</tbody>
</table>

Load/Store Dual, Half, Signed Byte (register)

These instructions are under Extra load/store.

<table>
<thead>
<tr>
<th>cond</th>
<th>op0</th>
<th>op2</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111 00 0 P U 0 W 01</td>
<td>Load/Store Dual, Half, Signed Byte (register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Load/Store Dual, Half, Signed Byte (register, literal)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: cond != 1111 && op2 != 00 && cond != 1111 && op2 != 00
<table>
<thead>
<tr>
<th>P</th>
<th>W</th>
<th>o1</th>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>STRH (register) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>LDRD (register) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>STRD (register) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>LDRH (register) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>LDRSB (register) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>LDRSH (register) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>STRHT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>01</td>
<td>LDRHT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>LDRSBT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>LDRSHT</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>STRH (register) — pre-indexed</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>LDRD (register) — pre-indexed</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>STRD (register) — pre-indexed</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>LDRH (register) — pre-indexed</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>LDRSB (register) — pre-indexed</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>LDRSH (register) — pre-indexed</td>
<td></td>
</tr>
</tbody>
</table>

**Load/Store Dual, Half, Signed Byte (immediate, literal)**

These instructions are under Extra load/store.

The following constraints also apply to this encoding: cond != 1111 && op2 != 00 && cond != 1111 && op2 != 00

<table>
<thead>
<tr>
<th>P:W</th>
<th>Decode fields</th>
<th>Rn</th>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>!= 01</td>
<td>1</td>
<td>1111</td>
<td>LDRD (literal)</td>
</tr>
<tr>
<td>0</td>
<td>!= 01</td>
<td>1</td>
<td>1111</td>
<td>LDRH (literal)</td>
</tr>
<tr>
<td>0</td>
<td>!= 01</td>
<td>1</td>
<td>1111</td>
<td>LDRSB (literal)</td>
</tr>
<tr>
<td>0</td>
<td>!= 01</td>
<td>1</td>
<td>1111</td>
<td>LDRSH (literal)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>LDRD (immediate) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>STRH (immediate) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>STRD (immediate) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>LDRH (immediate) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>LDRSB (immediate) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>LDRSH (immediate) — post-indexed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>STRHT</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>LDRHT</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>LDRSBT</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>LDRSHT</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>! 1</td>
<td>1111</td>
<td>LDRD (immediate) — offset</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>STRH (immediate) — offset</td>
</tr>
</tbody>
</table>
Multiply and Accumulate

These instructions are under Data-processing and miscellaneous instructions.

The following constraints also apply to this encoding: cond \(!= 1111 \&\& \text{cond} \neq 1111\)

Synchronization primitives and Load-Acquire/Store-Release

These instructions are under Data-processing and miscellaneous instructions.

Load/Store Exclusive and Load-Acquire/Store-Release

These instructions are under Synchronization primitives and Load-Acquire/Store-Release.
The following constraints also apply to this encoding: cond != 1111 & cond != 1111

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0 0 0</td>
<td>STL</td>
</tr>
<tr>
<td>00 0 0 0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 1 0 0</td>
<td>STLEX</td>
</tr>
<tr>
<td>00 0 1 1 1</td>
<td>STREX</td>
</tr>
<tr>
<td>00 1 0 0 0</td>
<td>LDA</td>
</tr>
<tr>
<td>00 1 0 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 1 1 0 0</td>
<td>LDAEX</td>
</tr>
<tr>
<td>00 1 1 1 1</td>
<td>LDREX</td>
</tr>
<tr>
<td>01 0 0 0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 0 1 0 0</td>
<td>STLEXD</td>
</tr>
<tr>
<td>01 0 1 1 1</td>
<td>STREXD</td>
</tr>
<tr>
<td>01 1 0 0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 1 1 0 0</td>
<td>LDAEXD</td>
</tr>
<tr>
<td>01 1 1 1 1</td>
<td>LDREXD</td>
</tr>
<tr>
<td>10 0 0 0 0</td>
<td>STLB</td>
</tr>
<tr>
<td>10 0 0 0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 0 1 0 0</td>
<td>STLEXB</td>
</tr>
<tr>
<td>10 0 1 1 1</td>
<td>STREXB</td>
</tr>
<tr>
<td>10 1 0 0 0</td>
<td>LDB</td>
</tr>
<tr>
<td>10 1 0 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 1 1 0 0</td>
<td>LDAEXB</td>
</tr>
<tr>
<td>10 1 1 1 1</td>
<td>LDREXB</td>
</tr>
<tr>
<td>11 0 0 0 0</td>
<td>STLH</td>
</tr>
<tr>
<td>11 0 0 0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 0 1 0 0</td>
<td>STLEXH</td>
</tr>
<tr>
<td>11 0 1 1 1</td>
<td>STREXH</td>
</tr>
<tr>
<td>11 1 0 0 0</td>
<td>LDAH</td>
</tr>
<tr>
<td>11 1 0 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 1 0 0</td>
<td>LDAEXH</td>
</tr>
<tr>
<td>11 1 1 1 1</td>
<td>LDREXH</td>
</tr>
</tbody>
</table>

### Miscellaneous

These instructions are under Data-processing and miscellaneous instructions.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0 1 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 1 0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 1 1 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 1 1 0 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
01 001  BX
01 010  BXI
01 011  BLX (register)
01 110  UNALLOCATED
10 001  UNALLOCATED
10 010  UNALLOCATED
10 011  UNALLOCATED
10 110  UNALLOCATED
11 001  CLZ
11 010  UNALLOCATED
11 011  UNALLOCATED
11 110  ERET
111  Exception Generation
000  Move special register (register)
100  Cyclic Redundancy Check
101  Integer Saturating Arithmetic

### Exception Generation

These instructions are under **Miscellaneous**.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>HLT</td>
</tr>
<tr>
<td>01</td>
<td>BKPT</td>
</tr>
<tr>
<td>10</td>
<td>HVC</td>
</tr>
<tr>
<td>11</td>
<td>SMC</td>
</tr>
</tbody>
</table>

### Move special register (register)

These instructions are under **Miscellaneous**.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>0</td>
<td>MRS</td>
</tr>
<tr>
<td>x0</td>
<td>1</td>
<td>MRS (Banked register)</td>
</tr>
<tr>
<td>x1</td>
<td>0</td>
<td>MSR (register)</td>
</tr>
<tr>
<td>x1</td>
<td>1</td>
<td>MSR (Banked register)</td>
</tr>
</tbody>
</table>
Cyclic Redundancy Check

These instructions are under Miscellaneous.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz C</td>
<td></td>
</tr>
<tr>
<td>00 0</td>
<td>CRC32 — CRC32B</td>
</tr>
<tr>
<td>00 1</td>
<td>CRC32C — CRC32CB</td>
</tr>
<tr>
<td>01 0</td>
<td>CRC32 — CRC32H</td>
</tr>
<tr>
<td>01 1</td>
<td>CRC32C — CRC32CH</td>
</tr>
<tr>
<td>10 0</td>
<td>CRC32 — CRC32W</td>
</tr>
<tr>
<td>10 1</td>
<td>CRC32C — CRC32CW</td>
</tr>
<tr>
<td>11</td>
<td>CONSTRAINED UNPREDICTABLE</td>
</tr>
</tbody>
</table>

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings.

Integer Saturating Arithmetic

These instructions are under Miscellaneous.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>OADD</td>
</tr>
<tr>
<td>01</td>
<td>QSUB</td>
</tr>
<tr>
<td>10</td>
<td>QDADD</td>
</tr>
<tr>
<td>11</td>
<td>QDSUB</td>
</tr>
</tbody>
</table>

Halfword Multiply and Accumulate

These instructions are under Data-processing and miscellaneous instructions.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc M N</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>SMLABB, SMLABT, SMLATB, SMLATT</td>
</tr>
<tr>
<td>01 0 0</td>
<td>SMLAWB, SMLAWT — SMLAWB</td>
</tr>
<tr>
<td>01 0 1</td>
<td>SMULWB, SMULWT — SMULWB</td>
</tr>
</tbody>
</table>
Data-processing register (immediate shift)

These instructions are under [Data-processing and miscellaneous instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 1 0</td>
<td>SMLAWB, SMLAWT → SMLAWT</td>
</tr>
<tr>
<td>01 1 1</td>
<td>SMULWB, SMULWT → SMULWT</td>
</tr>
<tr>
<td>10</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT</td>
</tr>
<tr>
<td>11</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT</td>
</tr>
</tbody>
</table>

Data-processing register (immediate shift)

These instructions are under [Data-processing and miscellaneous instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>Integer Data Processing (three register, immediate shift)</td>
</tr>
<tr>
<td>10 1</td>
<td>Integer Test and Compare (two register, immediate shift)</td>
</tr>
<tr>
<td>11</td>
<td>Logical Arithmetic (three register, immediate shift)</td>
</tr>
</tbody>
</table>

Integer Data Processing (three register, immediate shift)

These instructions are under [Data-processing and miscellaneous instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AND, ANDS (register)</td>
</tr>
<tr>
<td>001</td>
<td>EOR, EORS (register)</td>
</tr>
<tr>
<td>010 0 ! = 1101</td>
<td>SUB, SUBS (register) → SUB</td>
</tr>
<tr>
<td>010 1 ! = 1101</td>
<td>SUB, SUBS (SP minus register) → SUB</td>
</tr>
<tr>
<td>011</td>
<td>RSB, RSBS (register)</td>
</tr>
<tr>
<td>100 0 ! = 1101</td>
<td>ADD, ADDS (register) → ADD</td>
</tr>
<tr>
<td>100 1 ! = 1101</td>
<td>ADD, ADDS (SP plus register) → ADD</td>
</tr>
<tr>
<td>111</td>
<td>ADC, ADCS (register)</td>
</tr>
<tr>
<td>110</td>
<td>SBC, SBCS (register)</td>
</tr>
<tr>
<td>111</td>
<td>RSC, RCS (register)</td>
</tr>
</tbody>
</table>

Integer Test and Compare (two register, immediate shift)

These instructions are under [Data-processing and miscellaneous instructions](#).
Top-level encodings for A32

The following constraints also apply to this encoding: \( \text{cond} \neq 1111 \&\& \text{cond} \neq 1111 \)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>TST (register)</td>
</tr>
<tr>
<td>01</td>
<td>TEQ (register)</td>
</tr>
<tr>
<td>10</td>
<td>CMP (register)</td>
</tr>
<tr>
<td>11</td>
<td>CMN (register)</td>
</tr>
</tbody>
</table>

**Logical Arithmetic (three register, immediate shift)**

These instructions are under [Data-processing register (immediate shift)].

The following constraints also apply to this encoding: \( \text{cond} \neq 1111 \&\& \text{cond} \neq 1111 \)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ORR, ORRS (register)</td>
</tr>
<tr>
<td>01</td>
<td>MOV, MOVS (register)</td>
</tr>
<tr>
<td>10</td>
<td>BIC, BICS (register)</td>
</tr>
<tr>
<td>11</td>
<td>MVN, MVNS (register)</td>
</tr>
</tbody>
</table>

**Data-processing register (register shift)**

These instructions are under [Data-processing and miscellaneous instructions].

The following constraints also apply to this encoding: \( \text{op0:op1} \neq 100 \)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>Integer Data Processing (three register, register shift)</td>
</tr>
<tr>
<td>10 1</td>
<td>Integer Test and Compare (two register, register shift)</td>
</tr>
<tr>
<td>11</td>
<td>Logical Arithmetic (three register, register shift)</td>
</tr>
</tbody>
</table>

**Integer Data Processing (three register, register shift)**

These instructions are under [Data-processing register (register shift)].

The following constraints also apply to this encoding: \( \text{cond} \neq 1111 \&\& \text{cond} \neq 1111 \)
### Integer Test and Compare (two register, register shift)

These instructions are under **Data-processing register (register shift)**.

| 000 | AND, ANDS (register-shifted register) |
| 001 | EOR, EORS (register-shifted register) |
| 010 | SUB, SUBS (register-shifted register) |
| 011 | RSB, RSBS (register-shifted register) |
| 100 | ADD, ADDS (register-shifted register) |
| 101 | ADC, ADCS (register-shifted register) |
| 110 | SBC, SBCS (register-shifted register) |
| 111 | RSC, RSCS (register-shifted register) |

The following constraints also apply to this encoding: \( \text{cond} \neq 1111 \) \&\& \( \text{cond} \neq 1111 \)

### Logical Arithmetic (three register, register shift)

These instructions are under **Data-processing register (register shift)**.

| 00 | TST (register-shifted register) |
| 01 | TEQ (register-shifted register) |
| 10 | CMP (register-shifted register) |
| 11 | CMN (register-shifted register) |

The following constraints also apply to this encoding: \( \text{cond} \neq 1111 \) \&\& \( \text{cond} \neq 1111 \)

### Data-processing immediate

These instructions are under **Data-processing and miscellaneous instructions**.

<table>
<thead>
<tr>
<th>001</th>
<th>OP0</th>
<th>OP1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORR, ORRS (register-shifted register)</td>
<td>MOV, MOVS (register-shifted register)</td>
<td>BIC, BICS (register-shifted register)</td>
</tr>
<tr>
<td>0x</td>
<td>Instruction Details</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><strong>Move Halfword (immediate)</strong></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><strong>Move Special Register andHints (immediate)</strong></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><strong>Integer Test and Compare (one register and immediate)</strong></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td><strong>Logical Arithmetic (two register and immediate)</strong></td>
<td></td>
</tr>
</tbody>
</table>

## Integer Data Processing (two register and immediate)

These instructions are under [Data-processing immediate](#).

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>opc</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The following constraints also apply to this encoding: `cond != 1111 && cond != 1111`

<table>
<thead>
<tr>
<th>opc</th>
<th>S</th>
<th>Rn</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td></td>
<td>AND, ANDS (immediate)</td>
</tr>
<tr>
<td>001</td>
<td></td>
<td></td>
<td>EOR, EORS (immediate)</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>!= 111</td>
<td>SUB, SUBS (immediate) — SUB</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1101</td>
<td>SUB, SUBS (SP minus immediate) — SUB</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1111</td>
<td>ADR — A2</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>!= 1101</td>
<td>SUB, SUBS (immediate) — SUBS</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1101</td>
<td>SUB, SUBS (SP minus immediate) — SUBS</td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td>RSB, RSBS (immediate)</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>!= 111</td>
<td>ADD, ADDS (immediate) — ADD</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1101</td>
<td>ADD, ADDS (SP plus immediate) — ADD</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1111</td>
<td>ADR — A1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>!= 1101</td>
<td>ADD, ADDS (immediate) — ADDS</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1101</td>
<td>ADD, ADDS (SP plus immediate) — ADDS</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td>ADC, ADCS (immediate)</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td>SBC, SBCS (immediate)</td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td>RSC, RSCS (immediate)</td>
</tr>
</tbody>
</table>

## Move Halfword (immediate)

These instructions are under [Data-processing immediate](#).

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>H</td>
<td>0</td>
<td>0</td>
<td>imm4</td>
<td>Rd</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The following constraints also apply to this encoding: `cond != 1111 && cond != 1111`

<table>
<thead>
<tr>
<th>H</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MOV, MOVS (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>MOVT</td>
</tr>
</tbody>
</table>

## Move Special Register and Hints (immediate)

These instructions are under [Data-processing immediate](#).
The following constraints also apply to this encoding: \text{cond} \neq 1111 \&\& \text{cond} \neq 1111

Integer Test and Compare (one register and immediate)

These instructions are under \text{Data-processing immediate}.

Logical Arithmetic (two register and immediate)

These instructions are under \text{Data-processing immediate}.
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ORR, ORRS (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>MOV, MOVS (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>BIC, BICS (immediate)</td>
</tr>
<tr>
<td>11</td>
<td>MVN, MVNS (immediate)</td>
</tr>
</tbody>
</table>

#### Load/Store Word, Unsigned Byte (immediate, literal)

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

<table>
<thead>
<tr>
<th>P:W</th>
<th>Decode fields</th>
<th>Rn</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>o2 o1</td>
<td>Rn</td>
<td></td>
</tr>
<tr>
<td>!= 01</td>
<td>0 1</td>
<td>1111</td>
<td>LDR (literal)</td>
</tr>
<tr>
<td>!= 01</td>
<td>1 1</td>
<td>1111</td>
<td>LDRB (literal)</td>
</tr>
<tr>
<td>00</td>
<td>0 0</td>
<td>0 0 1111</td>
<td>STR (immediate) — post-indexed</td>
</tr>
<tr>
<td>00</td>
<td>0 1</td>
<td>1 1111</td>
<td>LDR (immediate) — post-indexed</td>
</tr>
<tr>
<td>00</td>
<td>1 0</td>
<td>0 1111</td>
<td>STRB (immediate) — post-indexed</td>
</tr>
<tr>
<td>00</td>
<td>1 1</td>
<td>1 1111</td>
<td>LDRB (immediate) — post-indexed</td>
</tr>
<tr>
<td>01</td>
<td>0 0</td>
<td>0</td>
<td>STRT</td>
</tr>
<tr>
<td>01</td>
<td>0 1</td>
<td>1</td>
<td>LDRT</td>
</tr>
<tr>
<td>01</td>
<td>1 0</td>
<td>0</td>
<td>STRBT</td>
</tr>
<tr>
<td>01</td>
<td>1 1</td>
<td>1</td>
<td>LDRBT</td>
</tr>
<tr>
<td>10</td>
<td>0 0</td>
<td>0 1111</td>
<td>STR (immediate) — offset</td>
</tr>
<tr>
<td>10</td>
<td>0 1</td>
<td>1 1111</td>
<td>LDR (immediate) — offset</td>
</tr>
<tr>
<td>10</td>
<td>1 0</td>
<td>0 1111</td>
<td>STRB (immediate) — offset</td>
</tr>
<tr>
<td>10</td>
<td>1 1</td>
<td>1 1111</td>
<td>LDRB (immediate) — offset</td>
</tr>
<tr>
<td>11</td>
<td>0 0</td>
<td>0</td>
<td>STR (immediate) — pre-indexed</td>
</tr>
<tr>
<td>11</td>
<td>0 1</td>
<td>1 1111</td>
<td>LDR (immediate) — pre-indexed</td>
</tr>
<tr>
<td>11</td>
<td>1 0</td>
<td>0</td>
<td>STRB (immediate) — pre-indexed</td>
</tr>
<tr>
<td>11</td>
<td>1 1</td>
<td>1 1111</td>
<td>LDRB (immediate) — pre-indexed</td>
</tr>
</tbody>
</table>

#### Load/Store Word, Unsigned Byte (register)

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

<table>
<thead>
<tr>
<th>P</th>
<th>Decode fields</th>
<th>Rn</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>o2 o1</td>
<td>Rn</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 0</td>
<td>0</td>
<td>STR (register) — post-indexed</td>
</tr>
<tr>
<td>P</td>
<td>o2</td>
<td>W</td>
<td>o1</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Media instructions

These instructions are under the top-level.

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

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xxx</td>
<td>Parallel Arithmetic</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>101</td>
<td>SEL</td>
</tr>
<tr>
<td>01000</td>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01000</td>
<td>xx0</td>
<td>PKHBT, PKHTB</td>
</tr>
<tr>
<td>01001</td>
<td>x01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01001</td>
<td>xx0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0110x</td>
<td>x01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0110x</td>
<td>xx0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x10</td>
<td>001</td>
<td>Saturate 16-bit</td>
</tr>
<tr>
<td>01x10</td>
<td>101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x11</td>
<td>x01</td>
<td>Reverse Bit/Byte</td>
</tr>
<tr>
<td>01x1x</td>
<td>xx0</td>
<td>Saturate 32-bit</td>
</tr>
<tr>
<td>01xxx</td>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01xxx</td>
<td>011</td>
<td>Extend and Add</td>
</tr>
<tr>
<td>10xxx</td>
<td>Signed multiply, Divide</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>000</td>
<td>Unsigned Sum of Absolute Differences</td>
</tr>
<tr>
<td>11000</td>
<td>100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11001</td>
<td>x00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1101x</td>
<td>x00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110xx</td>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1110x</td>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1110x</td>
<td>x00</td>
<td>Bitfield Insert</td>
</tr>
<tr>
<td>11110</td>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11111</td>
<td>111</td>
<td>Permanently UNDEFINED</td>
</tr>
<tr>
<td>1111x</td>
<td>x00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11x0x</td>
<td>x10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11x1x</td>
<td>x10</td>
<td>Bitfield Extract</td>
</tr>
</tbody>
</table>
Parallel Arithmetic

These instructions are under Media instructions.

The following constraints also apply to this encoding: \( \text{cond} \neq 1111 \) && \( \text{cond} \neq 1111 \)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>001</td>
<td>0 0 0</td>
</tr>
<tr>
<td>001</td>
<td>0 0 1</td>
</tr>
<tr>
<td>001</td>
<td>0 1 0</td>
</tr>
<tr>
<td>001</td>
<td>0 1 1</td>
</tr>
<tr>
<td>001</td>
<td>1 0 0</td>
</tr>
<tr>
<td>001</td>
<td>1 0 1</td>
</tr>
<tr>
<td>001</td>
<td>1 1 0</td>
</tr>
<tr>
<td>001</td>
<td>1 1 1</td>
</tr>
<tr>
<td>010</td>
<td>0 0 0</td>
</tr>
<tr>
<td>010</td>
<td>0 0 1</td>
</tr>
<tr>
<td>010</td>
<td>0 1 0</td>
</tr>
<tr>
<td>010</td>
<td>0 1 1</td>
</tr>
<tr>
<td>010</td>
<td>1 0 0</td>
</tr>
<tr>
<td>010</td>
<td>1 0 1</td>
</tr>
<tr>
<td>010</td>
<td>1 1 0</td>
</tr>
<tr>
<td>010</td>
<td>1 1 1</td>
</tr>
<tr>
<td>011</td>
<td>0 0 0</td>
</tr>
<tr>
<td>011</td>
<td>0 0 1</td>
</tr>
<tr>
<td>011</td>
<td>0 1 0</td>
</tr>
<tr>
<td>011</td>
<td>0 1 1</td>
</tr>
<tr>
<td>011</td>
<td>1 0 0</td>
</tr>
<tr>
<td>011</td>
<td>1 0 1</td>
</tr>
<tr>
<td>011</td>
<td>1 1 0</td>
</tr>
<tr>
<td>011</td>
<td>1 1 1</td>
</tr>
<tr>
<td>100</td>
<td>0 0 0</td>
</tr>
<tr>
<td>101</td>
<td>0 0 0</td>
</tr>
<tr>
<td>101</td>
<td>0 0 1</td>
</tr>
<tr>
<td>101</td>
<td>0 1 0</td>
</tr>
<tr>
<td>101</td>
<td>0 1 1</td>
</tr>
<tr>
<td>101</td>
<td>1 0 0</td>
</tr>
<tr>
<td>101</td>
<td>1 0 1</td>
</tr>
<tr>
<td>101</td>
<td>1 1 0</td>
</tr>
<tr>
<td>101</td>
<td>1 1 1</td>
</tr>
<tr>
<td>110</td>
<td>0 0 0</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>op1</td>
<td>B</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>

**Saturate 16-bit**

These instructions are under Media instructions.

```
| ! = 1111 | 0 | 1 | 0 | 1 | U | 1 | 0 | sat_imm | Rd | [1](1)(1)(1) | 0 | 0 | 1 | 1 | Rn |
```

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SSAT16</td>
</tr>
<tr>
<td>1</td>
<td>USAT16</td>
</tr>
</tbody>
</table>

**Reverse Bit/Byte**

These instructions are under Media instructions.

```
| ! = 1111 | 0 | 1 | 0 | 1 | o1 | 1 | 1 | (1)(1)(1)(1) | Rd | (1)(1)(1)(1) | o2 | 0 | 1 | 1 | Rm |
```

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>o2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Saturate 32-bit**

These instructions are under Media instructions.
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SSAT</td>
</tr>
<tr>
<td>1</td>
<td>USAT</td>
</tr>
</tbody>
</table>

### Extend and Add

These instructions are under Media instructions.

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

### Signed multiply, Divide

These instructions are under Media instructions.

The following constraints also apply to this encoding: cond != 1111 && cond != 1111
<table>
<thead>
<tr>
<th>op1</th>
<th>Decode fields</th>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1111</td>
<td>001</td>
<td>SMUAD, SMUADX — SMUADX</td>
</tr>
<tr>
<td>000</td>
<td>1111</td>
<td>010</td>
<td>SMUSD, SMUSDX — SMUSD</td>
</tr>
<tr>
<td>000</td>
<td>1111</td>
<td>011</td>
<td>SMUSD, SMUSDX — SMUSDX</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td></td>
<td>SDIV</td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td></td>
<td>SMLALD, SMLALDX — SMLALD</td>
</tr>
<tr>
<td>100</td>
<td>001</td>
<td></td>
<td>SMLALD, SMLALDX — SMLALDX</td>
</tr>
<tr>
<td>100</td>
<td>010</td>
<td></td>
<td>SMLSLD, SMLSLDX — SMLSLD</td>
</tr>
<tr>
<td>100</td>
<td>011</td>
<td></td>
<td>SMLSLD, SMLSLDX — SMLSLDX</td>
</tr>
<tr>
<td>100</td>
<td>1xx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101</td>
<td>!= 1111</td>
<td>000</td>
<td>SMMLA, SMMLAR — SMMLA</td>
</tr>
<tr>
<td>101</td>
<td>!= 1111</td>
<td>001</td>
<td>SMMLA, SMMLAR — SMMLAR</td>
</tr>
<tr>
<td>101</td>
<td>01x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101</td>
<td>10x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101</td>
<td>110</td>
<td></td>
<td>SMMLS, SMMLSR — SMMLS</td>
</tr>
<tr>
<td>101</td>
<td>111</td>
<td></td>
<td>SMMLS, SMMLSR — SMMLSR</td>
</tr>
<tr>
<td>101</td>
<td>1111</td>
<td>000</td>
<td>SMMUL, SMMULR — SMMUL</td>
</tr>
<tr>
<td>101</td>
<td>1111</td>
<td>001</td>
<td>SMMUL, SMMULR — SMMULR</td>
</tr>
<tr>
<td>11x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**Unsigned Sum of Absolute Differences**

These instructions are under Media instructions.

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

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

### Decode fields

<table>
<thead>
<tr>
<th>Ra</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
<tr>
<td>1111</td>
</tr>
</tbody>
</table>

**Bitfield Insert**

These instructions are under Media instructions.

<table>
<thead>
<tr>
<th>msb</th>
<th>Rd</th>
<th>lsb</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

### Decode fields

<table>
<thead>
<tr>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
</tr>
</tbody>
</table>
Permanently UNDEFINED

These instructions are under Media instructions.

```
<table>
<thead>
<tr>
<th></th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>BFC</td>
</tr>
</tbody>
</table>
```

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Bitfield Extract

These instructions are under Media instructions.

```
<table>
<thead>
<tr>
<th></th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1110</td>
<td>UDF</td>
</tr>
</tbody>
</table>
```

Branch, branch with link, and block data transfer

These instructions are under the top-level.

```
<table>
<thead>
<tr>
<th></th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SBFX</td>
</tr>
<tr>
<td>1</td>
<td>UBFX</td>
</tr>
</tbody>
</table>
```

Exception Save/Restore

These instructions are under Branch, branch with link, and block data transfer.

```
<table>
<thead>
<tr>
<th></th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0</td>
<td>Exception Save/Restore</td>
</tr>
<tr>
<td>0</td>
<td>Load/Store Multiple</td>
</tr>
<tr>
<td>1</td>
<td>Branch (immediate)</td>
</tr>
</tbody>
</table>
```

Page 1313
Load/Store Multiple

These instructions are under **Branch, branch with link, and block data transfer**.

The following constraints also apply to this encoding: \( \text{cond} \neq 1111 \) && \( \text{cond} \neq 1111 \)

Branch (immediate)

These instructions are under **Branch, branch with link, and block data transfer**.

System register access, Advanced SIMD, floating-point, and Supervisor call

These instructions are under the **top-level**.
# Supervisor call

These instructions are under [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

<table>
<thead>
<tr>
<th>cond</th>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>!= 1111</td>
<td>10 1x 1</td>
<td>Supervisor call</td>
</tr>
<tr>
<td>!= 1111</td>
<td>10 11 0</td>
<td></td>
</tr>
<tr>
<td>!= 1111</td>
<td>10 1x 1</td>
<td>Advanced SIMD and System register load/store and 64-bit move</td>
</tr>
</tbody>
</table>

---

# Unconditional Advanced SIMD and floating-point instructions

These instructions are under [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

<table>
<thead>
<tr>
<th>cond</th>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: op0<2:1> != 11

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td></td>
<td>0x</td>
<td>Advanced SIMD three registers of the same length extension</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>0 != 00</td>
<td>Floating-point minNum/maxNum</td>
</tr>
<tr>
<td>101</td>
<td>00</td>
<td>1 !&lt;00</td>
<td>Floating-point extraction and insertion</td>
</tr>
<tr>
<td>101</td>
<td>111</td>
<td>1 !&lt;00</td>
<td>Floating-point directed convert to integer</td>
</tr>
<tr>
<td>10x</td>
<td>00</td>
<td>Advanced SIMD and floating-point multiply with accumulate</td>
<td></td>
</tr>
<tr>
<td>10x</td>
<td>00</td>
<td>Advanced SIMD and floating-point dot product</td>
<td></td>
</tr>
</tbody>
</table>

---

# Advanced SIMD three registers of the same length extension

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#).
### Instruction Details

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>VCADD</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>64-bit SIMD vector</td>
<td>Armv8.3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>128-bit SIMD vector</td>
<td>Armv8.3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VMMLA</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VDOT (vector)</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VSMMLA</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>VUMMLA</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>VFMAL (vector)</td>
<td>Armv8.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VSDOT (vector)</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VUDOT (vector)</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>VFMA_BF (BFloat16, vector)</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VSMMLA</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VSDOT (vector)</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VUSDOT (vector)</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>VCMLA</td>
<td>Armv8.3</td>
</tr>
</tbody>
</table>

### Architecture Version

- Armv8.3
- Armv8.6
- Armv8.2

### Top-level encodings for A32

- VCADD: 64-bit SIMD vector
- VDOT: 64-bit SIMD vector
- VDOT: 128-bit SIMD vector
- VMMLA: 128-bit SIMD vector
- VUDOT: 128-bit SIMD vector
- VFMAL: 128-bit SIMD vector
- VSDOT: 128-bit SIMD vector
- VUSDOT: 128-bit SIMD vector
- VCMLA: 128-bit SIMD vector
- VFMAB, VFMAT (BFloat16, vector): 128-bit SIMD vector
### Floating-point minNum/maxNum

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#). The following constraints also apply to this encoding: size != 00 && size != 00

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VMAXNM</td>
</tr>
<tr>
<td>1</td>
<td>VMINNM</td>
</tr>
</tbody>
</table>

### Floating-point extraction and insertion

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#). The following constraints also apply to this encoding: size != 00 && size != 00

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 0</td>
<td>VMO VX</td>
</tr>
<tr>
<td>10 1</td>
<td>VINS</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Floating-point directed convert to integer

These instructions are under [Unconditional Advanced SIMD and floating-point instructions](#). The following constraints also apply to this encoding: size != 00 && size != 00

<table>
<thead>
<tr>
<th>o1</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00 00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>00 01</td>
<td>VRINTA (floating-point)</td>
</tr>
<tr>
<td>10</td>
<td>00 10</td>
<td>VRINTN (floating-point)</td>
</tr>
<tr>
<td>11</td>
<td>00 11</td>
<td>VRINTP (floating-point)</td>
</tr>
<tr>
<td>1</td>
<td>00 1</td>
<td>VRINTM (floating-point)</td>
</tr>
<tr>
<td>1</td>
<td>01 1</td>
<td>VCVTA (floating-point)</td>
</tr>
<tr>
<td>1</td>
<td>01 0</td>
<td>VCVTN (floating-point)</td>
</tr>
</tbody>
</table>
Advanced SIMD and floating-point multiply with accumulate

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

```
<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>Q</th>
<th>U</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point</td>
<td>Armv8.3</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td></td>
<td>VFMAVL (by scalar)</td>
<td>Armv8.2</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1</td>
<td></td>
<td>VFMSL (by scalar)</td>
<td>Armv8.2</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1</td>
<td></td>
<td>VFMAB, VFMAT (BFloat16, by scalar)</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point</td>
<td>Armv8.3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
```

Advanced SIMD and floating-point dot product

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

```
<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VDOT (by element) — 128-bit SIMD vector</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VSDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.2</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VUDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.2</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VSDOT (by element) — 128-bit SIMD vector</td>
<td>Armv8.2</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>VUDOT (by element) — 128-bit SIMD vector</td>
<td>Armv8.2</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VUSDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VUSDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VUSDOT (by element) — 128-bit SIMD vector</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>VUSDOT (by element) — 128-bit SIMD vector</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Advanced SIMD and System register load/store and 64-bit move

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>00x0</td>
<td>Advanced SIMD and floating-point 64-bit move</td>
</tr>
<tr>
<td>00x11</td>
<td>System register 64-bit move</td>
</tr>
<tr>
<td>!= 00x0</td>
<td>Advanced SIMD and floating-point load/store</td>
</tr>
<tr>
<td>!= 00x11</td>
<td>System register load/store</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Advanced SIMD and floating-point 64-bit move

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D op size opc2 o3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0x 00 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0 10 00 1</td>
<td>VMOV (between two general-purpose registers and two single-precision registers) — from general-purpose registers</td>
</tr>
<tr>
<td>1 0 11 00 1</td>
<td>VMOV (between two general-purpose registers and a doubleword floating-point register) — from general-purpose registers</td>
</tr>
<tr>
<td>1 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1 10 00 1</td>
<td>VMOV (between two general-purpose registers and two single-precision registers) — to general-purpose registers</td>
</tr>
<tr>
<td>1 1 11 00 1</td>
<td>VMOV (between two general-purpose registers and a doubleword floating-point register) — to general-purpose registers</td>
</tr>
</tbody>
</table>

System register 64-bit move

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D L</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0</td>
<td>MCRR</td>
</tr>
<tr>
<td>1 1</td>
<td>MRRC</td>
</tr>
</tbody>
</table>
## Advanced SIMD and floating-point load/store

These instructions are under **Advanced SIMD and System register load/store and 64-bit move**.

<table>
<thead>
<tr>
<th>P</th>
<th>U</th>
<th>W</th>
<th>L</th>
<th>Rn</th>
<th>size</th>
<th>imm8</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>VSTM, VSTMDB, VSTMIA</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>xxxxxxx0</td>
<td></td>
<td></td>
<td>VSTM, VSTMDB, VSTMIA</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>xxxxxxx1</td>
<td></td>
<td></td>
<td>FSTMDBX, FSTMIAX — Increment After</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>VLDM, VLDMDB, VLDMA</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>xxxxxxx0</td>
<td></td>
<td></td>
<td>VLDM, VLDMDB, VLDMA</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>xxxxxxx1</td>
<td></td>
<td></td>
<td>FLDM*X (FLDMDBX, FLDMIAX) — Increment After</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VSTR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>!= 1111</td>
<td>0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td></td>
<td></td>
<td>VSTM, VSTMDB, VSTMIA</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>xxxxxxx0</td>
<td></td>
<td></td>
<td>VSTM, VSTMDB, VSTMIA</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>xxxxxxx1</td>
<td></td>
<td></td>
<td>FSTMDBX, FSTMIAX — Decrement Before</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>VLDM, VLDMDB, VLDMA</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>xxxxxxx0</td>
<td></td>
<td></td>
<td>VLDM, VLDMDB, VLDMA</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>xxxxxxx1</td>
<td></td>
<td></td>
<td>FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td>VLDR (literal)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

## System register load/store

These instructions are under **Advanced SIMD and System register load/store and 64-bit move**.

<table>
<thead>
<tr>
<th>P</th>
<th>U:W</th>
<th>D</th>
<th>L</th>
<th>Rn</th>
<th>CRd</th>
<th>cp15</th>
<th>imm8</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>= 000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>= 000</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0101</td>
<td>0</td>
<td></td>
<td></td>
<td>LDC (literal)</td>
</tr>
<tr>
<td>= 000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0101</td>
<td>0</td>
<td></td>
<td>LDC (immediate) — post-indexed</td>
</tr>
<tr>
<td>0x1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0101</td>
<td>0</td>
<td></td>
<td></td>
<td>STC — post-indexed</td>
</tr>
<tr>
<td>0x1</td>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>0101</td>
<td>0</td>
<td></td>
<td></td>
<td>LDC (immediate) — post-indexed</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0101</td>
<td>0</td>
<td></td>
<td></td>
<td>STC — unindexed</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>0101</td>
<td>0</td>
<td></td>
<td></td>
<td>LDC (immediate) — unindexed</td>
</tr>
<tr>
<td>1x0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0101</td>
<td>0</td>
<td></td>
<td></td>
<td>STC — offset</td>
</tr>
</tbody>
</table>
### Advanced SIMD and System register 32-bit move

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
<th>Architecture version</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>000 001</td>
<td>VMOV (between general-purpose register and half-precision)</td>
<td>Armv8.2</td>
</tr>
<tr>
<td>000 010</td>
<td>VMOV (between general-purpose register and single-precision)</td>
<td>-</td>
</tr>
<tr>
<td>001 010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x 010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10x 010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>110 010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>111 010</td>
<td>Floating-point move special register</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>Advanced SIMD 8/16/32-bit element move/duplicate</td>
<td>-</td>
</tr>
<tr>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11x</td>
<td>System register 32-bit move</td>
<td>-</td>
</tr>
</tbody>
</table>

### Floating-point move special register

These instructions are under Advanced SIMD and System register 32-bit move.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture version</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VMSR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>VMRS</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced SIMD 8/16/32-bit element move/duplicate

These instructions are under Advanced SIMD and System register 32-bit move.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture version</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VMSR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>VMRS</td>
<td></td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: cond != 1111 & cond != 1111
Top-level encodings for A32

System register 32-bit move

These instructions are under Advanced SIMD and System register 32-bit move.

Floating-point data-processing

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

Floating-point data-processing (two registers)

These instructions are under Floating-point data-processing.

Floating-point data-processing (three registers)

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

The following constraints also apply to this encoding: cond != 1111 && cond != 1111
### Top-level encodings for A32

<table>
<thead>
<tr>
<th>o1</th>
<th>Decode fields</th>
<th>o3</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>010</td>
<td>0</td>
<td>VCVTB — half-precision to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>01</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>1</td>
<td>VCVTT — half-precision to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>01</td>
<td>VCVTB (BF16)</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>01</td>
<td>VCVTT (BF16)</td>
<td>Armv8.6</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>10</td>
<td>VCVTB — single-precision to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>11</td>
<td>VCVTT — single-precision to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>11</td>
<td>VCVTT — double-precision to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
<td>VCMP — A1</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1</td>
<td>VCMPE — A1</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>0</td>
<td>VCMP — A2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>1</td>
<td>VCMPE — A2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>0</td>
<td>VRINTR</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>1</td>
<td>VRINTZ (floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>0</td>
<td>VRINTX (floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>01</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>10</td>
<td>VCVT (between double-precision and single-precision) — single-precision to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>11</td>
<td>VCVT (between double-precision and single-precision) — double-precision to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td></td>
<td>VCVT (integer to floating-point, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>01</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>11</td>
<td>VCVT</td>
<td>Armv8.3</td>
</tr>
<tr>
<td>1</td>
<td>01x</td>
<td></td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>0</td>
<td>VCVTR</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>0</td>
<td>VCVTR</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11x</td>
<td></td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>-</td>
</tr>
</tbody>
</table>

### Floating-point move move immediate

These instructions are under **Floating-point data-processing**.

The following constraints also apply to this encoding: cond != 1111 && cond != 1111
Floating-point data-processing (three registers)

These instructions are under Floating-point data-processing.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>o0:o1 size o2</td>
<td>! = 1111 00 UNALLOCATED</td>
</tr>
<tr>
<td>000</td>
<td>0 VMLA (floating-point)</td>
</tr>
<tr>
<td>000</td>
<td>1 VMLS (floating-point)</td>
</tr>
<tr>
<td>001</td>
<td>0 VNMLS</td>
</tr>
<tr>
<td>001</td>
<td>1 VNMLA</td>
</tr>
<tr>
<td>010</td>
<td>0 VMUL (floating-point)</td>
</tr>
<tr>
<td>010</td>
<td>1 VNMUL</td>
</tr>
<tr>
<td>011</td>
<td>0 VADD (floating-point)</td>
</tr>
<tr>
<td>011</td>
<td>1 VSUB (floating-point)</td>
</tr>
<tr>
<td>100</td>
<td>0 VDIV</td>
</tr>
<tr>
<td>101</td>
<td>0 VFNMS</td>
</tr>
<tr>
<td>101</td>
<td>1 VFNMA</td>
</tr>
<tr>
<td>110</td>
<td>0 VFMA</td>
</tr>
<tr>
<td>110</td>
<td>1 VFMS</td>
</tr>
</tbody>
</table>

Unconditional instructions

These instructions are under top-level.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td>! = 111100 Miscellaneous</td>
</tr>
<tr>
<td>00x</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>01x</td>
<td>Advanced SIMD data-processing</td>
</tr>
<tr>
<td>1xx</td>
<td>Memory hints and barriers</td>
</tr>
<tr>
<td>100</td>
<td>Advanced SIMD element or structure load/store</td>
</tr>
<tr>
<td>101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Miscellaneous

These instructions are under Unconditional instructions.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
<th>Architecture version</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td>! = 111000 UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>Change Process State</td>
<td>-</td>
</tr>
</tbody>
</table>
### Top-level encodings for A32

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10001 x100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10001 xx01</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10001 0000</td>
<td><strong>SETPAN</strong> Armv8.1</td>
<td></td>
</tr>
<tr>
<td>1000 x 0111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10010 0111</td>
<td><strong>CONSTRAINED UNPREDICTABLE</strong></td>
<td>-</td>
</tr>
<tr>
<td>10011 0111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1001 x xx0x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>100xx 0011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>100xx 0x10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>100xx 1x1x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>101xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

The behavior of the **CONSTRAINED UNPREDICTABLE** encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#).

### Change Process State

These instructions are under **Miscellaneous**.

<table>
<thead>
<tr>
<th>Imod</th>
<th>M</th>
<th>Op</th>
<th>I</th>
<th>F</th>
<th>Mode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0xxx</td>
<td>SETEND</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD data-processing

These instructions are under **Unconditional instructions**.

<table>
<thead>
<tr>
<th>Code</th>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111001</td>
<td></td>
<td></td>
<td>Advanced SIMD three registers of the same length</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Advanced SIMD three registers of the same length</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Advanced SIMD two registers, or three registers of different lengths</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Advanced SIMD shifts and immediate generation</td>
</tr>
</tbody>
</table>

### Advanced SIMD three registers of the same length

These instructions are under **Advanced SIMD data-processing**.

<table>
<thead>
<tr>
<th>Code</th>
<th>Imod</th>
<th>M</th>
<th>Op</th>
<th>I</th>
<th>F</th>
<th>Mode</th>
<th>Vn</th>
<th>Vd</th>
<th>opc</th>
<th>N</th>
<th>Q</th>
<th>M</th>
<th>o1</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td></td>
<td></td>
<td>o1</td>
<td>Vm</td>
</tr>
<tr>
<td>U</td>
<td>Decode fields</td>
<td>opc</td>
<td>Q</td>
<td>o1</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---------------</td>
<td>-----</td>
<td>---</td>
<td>-----</td>
<td>----------------------</td>
<td>----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 1100</td>
<td>1</td>
<td></td>
<td></td>
<td>VFMA</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 1101</td>
<td>0</td>
<td></td>
<td></td>
<td>VADD (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 1101</td>
<td>1</td>
<td></td>
<td></td>
<td>VMLA (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 1110</td>
<td>0</td>
<td></td>
<td></td>
<td>VCEQ (register) — A2</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 1111</td>
<td>0</td>
<td></td>
<td></td>
<td>VMAX (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 1111</td>
<td>1</td>
<td></td>
<td></td>
<td>VRECPS</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0</td>
<td></td>
<td></td>
<td>VHADD</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 0001</td>
<td>1</td>
<td></td>
<td></td>
<td>VAND (register)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>1</td>
<td></td>
<td></td>
<td>VOADD</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>0</td>
<td></td>
<td></td>
<td>VRHADD</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 1100</td>
<td>0</td>
<td></td>
<td></td>
<td>SHA1C</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>0</td>
<td></td>
<td></td>
<td>VHSUB</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01 0001</td>
<td>1</td>
<td></td>
<td></td>
<td>VBIC (register)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>1</td>
<td></td>
<td></td>
<td>VOSUB</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>0</td>
<td></td>
<td></td>
<td>VCGT (register) — A1</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>1</td>
<td></td>
<td></td>
<td>VCGE (register) — A1</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01 1100</td>
<td>0</td>
<td></td>
<td></td>
<td>SHA1P</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 1100</td>
<td>1</td>
<td></td>
<td></td>
<td>VFMS</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 1101</td>
<td>0</td>
<td></td>
<td></td>
<td>VSUB (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 1101</td>
<td>1</td>
<td></td>
<td></td>
<td>VMLS (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 1110</td>
<td>0</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 1111</td>
<td>0</td>
<td></td>
<td></td>
<td>VMIN (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 1111</td>
<td>1</td>
<td></td>
<td></td>
<td>VRSORTS</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td>0</td>
<td></td>
<td></td>
<td>VSHL (register)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>0</td>
<td></td>
<td></td>
<td>VADD (integer)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 0001</td>
<td>1</td>
<td></td>
<td></td>
<td>VORR (register)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>1</td>
<td></td>
<td></td>
<td>VTST</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td>1</td>
<td></td>
<td></td>
<td>VOSHL (register)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td>0</td>
<td></td>
<td></td>
<td>VMLA (integer)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>0</td>
<td></td>
<td></td>
<td>VRSHL</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>1</td>
<td></td>
<td></td>
<td>VORSHL</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>0</td>
<td></td>
<td></td>
<td>VODMULH</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 1100</td>
<td>0</td>
<td></td>
<td></td>
<td>SHA1M</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 1111</td>
<td>1</td>
<td></td>
<td></td>
<td>VPADD (integer)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>0</td>
<td></td>
<td></td>
<td>VMAX (integer)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 0001</td>
<td>1</td>
<td></td>
<td></td>
<td>VORN (register)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>1</td>
<td></td>
<td></td>
<td>VMIN (integer)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>0</td>
<td></td>
<td></td>
<td>VABD (integer)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>1</td>
<td></td>
<td></td>
<td>VABA</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 1100</td>
<td>0</td>
<td></td>
<td></td>
<td>SHA1SU0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 1101</td>
<td>0</td>
<td></td>
<td></td>
<td>VPADD (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 1101</td>
<td>1</td>
<td></td>
<td></td>
<td>VMUL (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 1110</td>
<td>0</td>
<td></td>
<td></td>
<td>VCGE (register) — A2</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 1110</td>
<td>1</td>
<td></td>
<td></td>
<td>VACGE</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 1111</td>
<td>0</td>
<td>0</td>
<td></td>
<td>VPMAX (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 1111</td>
<td>1</td>
<td></td>
<td></td>
<td>VMAXNM</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00 0001</td>
<td>1</td>
<td></td>
<td></td>
<td>VEOR</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD two registers, or three registers of different lengths

These instructions are under Advanced SIMD data-processing.

#### Advanced SIMD two registers misc

These instructions are under Advanced SIMD two registers, or three registers of different lengths.
<table>
<thead>
<tr>
<th>size</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0010</td>
<td>VREV16</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 0011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 010x</td>
<td>VPADDL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 0110 0</td>
<td>AESE</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 0110 1</td>
<td>AESD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 0111 0</td>
<td>AESMC</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 0111 1</td>
<td>AESIMC</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 1000</td>
<td>VCLS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 1000 0</td>
<td>VSWP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 1001</td>
<td>VCLZ</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 1010</td>
<td>VCNT</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 1011</td>
<td>VMVN (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 1100 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 110x</td>
<td>VPADAL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 1110</td>
<td>VQABS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>00 1111</td>
<td>VONEG</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 x000</td>
<td>VCGT (immediate #0)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 x001</td>
<td>VCGE (immediate #0)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 x010</td>
<td>VCEO (immediate #0)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 x011</td>
<td>VCLE (immediate #0)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 x100</td>
<td>VCLT (immediate #0)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 x110</td>
<td>VABS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 x111</td>
<td>VNEG</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 0101 1</td>
<td>SHA1H</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 10 1100 1</td>
<td>VCVT (from single-precision to BFloat16, Advanced SIMD)</td>
<td>Armv8.6</td>
<td></td>
</tr>
<tr>
<td>10 0001</td>
<td>VTRN</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 0010</td>
<td>VUZP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 0011</td>
<td>VZIP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 0100 0</td>
<td>VMOVN</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 0100 1</td>
<td>VQMOVN, VQMOVUN — VQMOVUN</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 0101</td>
<td>VQMOVN, VQMOVUN — VQMOVN</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 0110 0</td>
<td>VSHLL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 0111 0</td>
<td>SHA1SU1</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 0111 1</td>
<td>SHA256SU0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 1000</td>
<td>VREINTN (Advanced SIMD)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 1001</td>
<td>VREINTX (Advanced SIMD)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 1010</td>
<td>VREINTA (Advanced SIMD)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 1011</td>
<td>VREINTZ (Advanced SIMD)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 10 1100 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 1100 0</td>
<td>VCVT (between half-precision and single-precision, Advanced SIMD) — single-precision to half-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 1101</td>
<td>VREINTM (Advanced SIMD)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 1110 0</td>
<td>VCVT (between half-precision and single-precision, Advanced SIMD) — half-precision to single-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 1110 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10 1111</td>
<td>VREINTP (Advanced SIMD)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11 000x</td>
<td>VCVTA (Advanced SIMD)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11 001x</td>
<td>VCVTN (Advanced SIMD)</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD duplicate (scalar)

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>VDUP (scalar)</td>
</tr>
<tr>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD three registers of different lengths

These instructions are under [Advanced SIMD two registers, or three registers of different lengths](#).

The following constraints also apply to this encoding: size != 11 && size != 11
Advanced SIMD two registers and a scalar

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

The following constraints also apply to this encoding: size != 11 && size != 11

Advanced SIMD shifts and immediate generation

These instructions are under Advanced SIMD data-processing.

Advanced SIMD one register and modified immediate

These instructions are under Advanced SIMD shifts and immediate generation.
### Decode fields

<table>
<thead>
<tr>
<th>cmode</th>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx0</td>
<td>0</td>
<td>VMOV (immediate) — A1</td>
</tr>
<tr>
<td>0xx0</td>
<td>1</td>
<td>VMVN (immediate) — A1</td>
</tr>
<tr>
<td>0xx1</td>
<td>0</td>
<td>VORR (immediate) — A1</td>
</tr>
<tr>
<td>0xx1</td>
<td>1</td>
<td>VBIC (immediate) — A1</td>
</tr>
<tr>
<td>10x0</td>
<td>0</td>
<td>VMOV (immediate) — A3</td>
</tr>
<tr>
<td>10x0</td>
<td>1</td>
<td>VMVN (immediate) — A2</td>
</tr>
<tr>
<td>10x1</td>
<td>0</td>
<td>VORR (immediate) — A2</td>
</tr>
<tr>
<td>10x1</td>
<td>1</td>
<td>VBIC (immediate) — A2</td>
</tr>
<tr>
<td>11xx</td>
<td>0</td>
<td>VMOV (immediate) — A4</td>
</tr>
<tr>
<td>110x</td>
<td>1</td>
<td>VMVN (immediate) — A3</td>
</tr>
<tr>
<td>1110</td>
<td>1</td>
<td>VMOV (immediate) — A5</td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD two registers and shift amount

These instructions are under Advanced SIMD shifts and immediate generation.

The following constraints also apply to this encoding: imm3H:imm3L:Vd:opc:L != 000xxxxxxxxxxx0

### Memory hints and barriers

These instructions are under Unconditional instructions.
### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xx1</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>Barriers</td>
<td></td>
</tr>
<tr>
<td>011x1</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td></td>
</tr>
<tr>
<td>0xxx0</td>
<td>Preload (immediate)</td>
<td></td>
</tr>
<tr>
<td>1xxx0</td>
<td>0</td>
<td>Preload (register)</td>
</tr>
<tr>
<td>1xxx1</td>
<td>0</td>
<td>CONSTRAINED UNPREDICTABLE</td>
</tr>
<tr>
<td>1xxxx</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings*.

### Barriers

These instructions are under [Memory hints and barriers](#).

### Preload (immediate)

These instructions are under [Memory hints and barriers](#).

---

**Top-level encodings for A32**

---

Page 1332
## Preload (register)

These instructions are under **Memory hints and barriers**.

<table>
<thead>
<tr>
<th>D</th>
<th>o2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Reserved hint, behaves as NOP</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>PLI (register)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>PLD, PLDW (register) — preload write</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>PLD, PLDW (register) — preload read</td>
</tr>
</tbody>
</table>

## Advanced SIMD element or structure load/store

These instructions are under **Unconditional instructions**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>!= 11</td>
</tr>
</tbody>
</table>

## Advanced SIMD load/store multiple structures

These instructions are under **Advanced SIMD element or structure load/store**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>itype</td>
</tr>
<tr>
<td>0</td>
<td>000x</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
</tr>
<tr>
<td>0</td>
<td>010x</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
</tr>
<tr>
<td>0</td>
<td>100x</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>000x</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
</tr>
<tr>
<td>1</td>
<td>010x</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>100x</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
</tr>
</tbody>
</table>
**Advanced SIMD load single structure to all lanes**

These instructions are under [Advanced SIMD element or structure load/store](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 00</td>
<td>VLD1 (single element to all lanes)</td>
</tr>
<tr>
<td>1 01</td>
<td>VLD2 (single 2-element structure to all lanes)</td>
</tr>
<tr>
<td>1 10 0</td>
<td>VLD3 (single 3-element structure to all lanes)</td>
</tr>
<tr>
<td>1 10 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 11</td>
<td>VLD4 (single 4-element structure to all lanes)</td>
</tr>
</tbody>
</table>

**Advanced SIMD load/store single structure to one lane**

These instructions are under [Advanced SIMD element or structure load/store](#).

The following constraints also apply to this encoding: size != 11 && size != 11

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00 00</td>
<td>VST1 (single element from one lane) — A1</td>
</tr>
<tr>
<td>0 00 01</td>
<td>VST2 (single 2-element structure from one lane) — A1</td>
</tr>
<tr>
<td>0 00 10</td>
<td>VST3 (single 3-element structure from one lane) — A1</td>
</tr>
<tr>
<td>0 00 11</td>
<td>VST4 (single 4-element structure from one lane) — A1</td>
</tr>
<tr>
<td>0 01 00</td>
<td>VST1 (single element from one lane) — A2</td>
</tr>
<tr>
<td>0 01 01</td>
<td>VST2 (single 2-element structure from one lane) — A2</td>
</tr>
<tr>
<td>0 01 10</td>
<td>VST3 (single 3-element structure from one lane) — A2</td>
</tr>
<tr>
<td>0 01 11</td>
<td>VST4 (single 4-element structure from one lane) — A2</td>
</tr>
<tr>
<td>0 10 00</td>
<td>VST1 (single element from one lane) — A3</td>
</tr>
<tr>
<td>0 10 01</td>
<td>VST2 (single 2-element structure from one lane) — A3</td>
</tr>
<tr>
<td>0 10 10</td>
<td>VST3 (single 3-element structure from one lane) — A3</td>
</tr>
<tr>
<td>0 10 11</td>
<td>VST4 (single 4-element structure from one lane) — A3</td>
</tr>
<tr>
<td>1 00 00</td>
<td>VLD1 (single element to one lane) — A1</td>
</tr>
<tr>
<td>1 00 01</td>
<td>VLD2 (single 2-element structure to one lane) — A1</td>
</tr>
<tr>
<td>1 00 10</td>
<td>VLD3 (single 3-element structure to one lane) — A1</td>
</tr>
<tr>
<td>1 00 11</td>
<td>VLD4 (single 4-element structure to one lane) — A1</td>
</tr>
<tr>
<td>1 01 00</td>
<td>VLD1 (single element to one lane) — A2</td>
</tr>
<tr>
<td>1 01 01</td>
<td>VLD2 (single 2-element structure to one lane) — A2</td>
</tr>
<tr>
<td>1 01 10</td>
<td>VLD3 (single 3-element structure to one lane) — A2</td>
</tr>
<tr>
<td>1 01 11</td>
<td>VLD4 (single 4-element structure to one lane) — A2</td>
</tr>
<tr>
<td>1 10 00</td>
<td>VLD1 (single element to one lane) — A3</td>
</tr>
<tr>
<td>1 10 01</td>
<td>VLD2 (single 2-element structure to one lane) — A3</td>
</tr>
<tr>
<td>1 10 10</td>
<td>VLD3 (single 3-element structure to one lane) — A3</td>
</tr>
<tr>
<td>1 10 11</td>
<td>VLD4 (single 4-element structure to one lane) — A3</td>
</tr>
</tbody>
</table>
Top-level encodings for T32

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>!= 111</td>
<td>16-bit</td>
</tr>
<tr>
<td>111 00</td>
<td>B — T2</td>
</tr>
<tr>
<td>111 != 00</td>
<td>32-bit</td>
</tr>
</tbody>
</table>

16-bit

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>00xxxx</td>
<td>Shift (immediate), add, subtract, move, and compare</td>
</tr>
<tr>
<td>010000</td>
<td>Data-processing (two low registers)</td>
</tr>
<tr>
<td>010001</td>
<td>Special data instructions and branch and exchange</td>
</tr>
<tr>
<td>01001x</td>
<td>LDR (literal) — T1</td>
</tr>
<tr>
<td>0101xx</td>
<td>Load/store (register offset)</td>
</tr>
<tr>
<td>011xxx</td>
<td>Load/store word/byte (immediate offset)</td>
</tr>
<tr>
<td>1000xx</td>
<td>Load/store halfword (immediate offset)</td>
</tr>
<tr>
<td>1001xx</td>
<td>Load/store (SP-relative)</td>
</tr>
<tr>
<td>1010xx</td>
<td>Add PC/SP (immediate)</td>
</tr>
<tr>
<td>1011xx</td>
<td>Miscellaneous 16-bit instructions</td>
</tr>
<tr>
<td>1100xx</td>
<td>Load/store multiple</td>
</tr>
<tr>
<td>1101xx</td>
<td>Conditional branch, and Supervisor Call</td>
</tr>
</tbody>
</table>

Shift (immediate), add, subtract, move, and compare

These instructions are under 16-bit.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2</td>
<td></td>
</tr>
<tr>
<td>0 11 0</td>
<td>Add, subtract (three low registers)</td>
</tr>
<tr>
<td>0 11 1</td>
<td>Add, subtract (two low registers and immediate)</td>
</tr>
<tr>
<td>0 != 11</td>
<td>MOV, MOVs (register) — T2</td>
</tr>
<tr>
<td>1</td>
<td>Add, subtract, compare, move (one low register and immediate)</td>
</tr>
</tbody>
</table>

Add, subtract (three low registers)

These instructions are under Shift (immediate), add, subtract, move, and compare.
Top-level encodings for T32

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 0</td>
<td>ADD, ADDS (register)</td>
</tr>
<tr>
<td>S 1</td>
<td>SUB, SUBS (register)</td>
</tr>
</tbody>
</table>

Add, subtract (two low registers and immediate)

These instructions are under Shift (immediate), add, subtract, move, and compare.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 0</td>
<td>ADD, ADDS (immediate)</td>
</tr>
<tr>
<td>S 1</td>
<td>SUB, SUBS (immediate)</td>
</tr>
</tbody>
</table>

Add, subtract, compare, move (one low register and immediate)

These instructions are under Shift (immediate), add, subtract, move, and compare.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op 00</td>
<td>MOV, MOVS (immediate)</td>
</tr>
<tr>
<td>op 01</td>
<td>CMP (immediate)</td>
</tr>
<tr>
<td>op 10</td>
<td>ADD, ADDS (immediate)</td>
</tr>
<tr>
<td>op 11</td>
<td>SUB, SUBS (immediate)</td>
</tr>
</tbody>
</table>

Data-processing (two low registers)

These instructions are under 16-bit.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op 0000</td>
<td>AND, ANDS (register)</td>
</tr>
<tr>
<td>op 0001</td>
<td>EOR, EORS (register)</td>
</tr>
<tr>
<td>op 0010</td>
<td>MOV, MOVS (register-shifted register) — logical shift left</td>
</tr>
<tr>
<td>op 0011</td>
<td>MOV, MOVS (register-shifted register) — logical shift right</td>
</tr>
<tr>
<td>op 0100</td>
<td>MOV, MOVS (register-shifted register) — arithmetic shift right</td>
</tr>
<tr>
<td>op 0101</td>
<td>ADC, ADCS (register)</td>
</tr>
<tr>
<td>op 0110</td>
<td>SBC, SBCS (register)</td>
</tr>
<tr>
<td>op 0111</td>
<td>MOV, MOVS (register-shifted register) — rotate right</td>
</tr>
<tr>
<td>1000</td>
<td>TST (register)</td>
</tr>
<tr>
<td>1001</td>
<td>RSB, RSBS (immediate)</td>
</tr>
<tr>
<td>1010</td>
<td>CMP (register)</td>
</tr>
<tr>
<td>1011</td>
<td>CMN (register)</td>
</tr>
</tbody>
</table>
### Special data instructions and branch and exchange

These instructions are under 16-bit.

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>010001</th>
<th>op0</th>
</tr>
</thead>
</table>

#### Branch and exchange

These instructions are under Special data instructions and branch and exchange.

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 1 0 0 0 1 1 1</th>
<th>L</th>
<th>Rm</th>
<th>(0)(0)(0)</th>
</tr>
</thead>
</table>

#### Add, subtract, compare, move (two high registers)

These instructions are under Special data instructions and branch and exchange.

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 1 0 0 0 1 != 11</th>
<th>D</th>
<th>Rs</th>
<th>Rd</th>
</tr>
</thead>
</table>

The following constraints also apply to this encoding: op != 11 && op != 11

| 00 1101 != 1101 | ADD, ADDS (register) |
| 00 1101       | ADD, ADDS (SP plus register) — T1 |
| 00 1101 != 1101 | ADD, ADDS (SP plus register) — T2 |
| 01             | CMP (register)       |
| 10             | MOV, MOVS (register) |

### Load/store (register offset)

These instructions are under 16-bit.

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 1 0 1 | L | B | H | Rm | Rn | Rt |
|----------------------------------------|--------|----|----|----|----|----|----|----|
Load/store word/byte (immediate offset)

These instructions are under 16-bit.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Load/store halfword (immediate offset)

These instructions are under 16-bit.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>imm5</td>
</tr>
<tr>
<td>0</td>
<td>STR (Immediate)</td>
</tr>
<tr>
<td>1</td>
<td>LDR (Immediate)</td>
</tr>
</tbody>
</table>

Load/store (SP-relative)

These instructions are under 16-bit.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>imm8</td>
</tr>
<tr>
<td>0</td>
<td>STRH (Immediate)</td>
</tr>
<tr>
<td>1</td>
<td>LDRH (Immediate)</td>
</tr>
</tbody>
</table>

Add PC/SP (Immediate)

These instructions are under 16-bit.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>imm8</td>
</tr>
<tr>
<td>0</td>
<td>STR (Immediate)</td>
</tr>
<tr>
<td>1</td>
<td>LDR (Immediate)</td>
</tr>
</tbody>
</table>
### Miscellaneous 16-bit instructions

These instructions are under [16-bit](#).

```
1514131211109876543210
1011 | op0 | op1 | op2 | op3
```

### Adjust SP (immediate)

These instructions are under [Miscellaneous 16-bit instructions](#).

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 1 0 0 0 0 S imm7
```

### Extend

These instructions are under [Miscellaneous 16-bit instructions](#).
**Change Processor State**

These instructions are under [Miscellaneous 16-bit instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>op</td>
<td>flags</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reverse bytes**

These instructions are under [Miscellaneous 16-bit instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>flags</td>
</tr>
<tr>
<td>0</td>
<td>SETEND</td>
</tr>
<tr>
<td>1</td>
<td>CPS, CPSID, CPSIE</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: \( \text{op} \neq 10 \) \&\& \( \text{op} \neq 10 \)

**Hints**

These instructions are under [Miscellaneous 16-bit instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>hint</td>
<td>0000</td>
</tr>
<tr>
<td>hint</td>
<td>0001</td>
</tr>
<tr>
<td>hint</td>
<td>0010</td>
</tr>
<tr>
<td>hint</td>
<td>0011</td>
</tr>
<tr>
<td>hint</td>
<td>0100</td>
</tr>
<tr>
<td>hint</td>
<td>0101</td>
</tr>
<tr>
<td>hint</td>
<td>011x</td>
</tr>
<tr>
<td>hint</td>
<td>1xxx</td>
</tr>
</tbody>
</table>

**Push and Pop**

These instructions are under [Miscellaneous 16-bit instructions](#).
Load/store multiple

These instructions are under 16-bit.

Conditional branch, and Supervisor Call

These instructions are under 16-bit.

Exception generation

These instructions are under Conditional branch, and Supervisor Call.

32-bit

These instructions are under the top-level.

The following constraints also apply to this encoding: op0<3:2> != 00
### System register access, Advanced SIMD, and floating-point

These instructions are under [32-bit](#).

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>111 op0 11 op1</td>
<td><strong>Advanced SIMD data-processing</strong></td>
</tr>
<tr>
<td>111 op0 1111 op0</td>
<td><strong>Additional Advanced SIMD and floating-point instructions</strong></td>
</tr>
</tbody>
</table>

### Advanced SIMD data-processing

These instructions are under [System register access, Advanced SIMD, and floating-point](#).

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td><strong>Advanced SIMD three registers of the same length</strong></td>
</tr>
<tr>
<td>1</td>
<td><strong>Advanced SIMD two registers, or three registers of different lengths</strong></td>
</tr>
<tr>
<td>1</td>
<td><strong>Advanced SIMD shifts and immediate generation</strong></td>
</tr>
</tbody>
</table>

---

### Top-level encodings for T32

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 xx0xx</td>
<td>Load/store multiple</td>
</tr>
<tr>
<td>0100 xx1xx</td>
<td>Load/store dual, load/store exclusive, load-acquire/store-release, and table branch</td>
</tr>
<tr>
<td>0101</td>
<td>Data-processing (shifted register)</td>
</tr>
<tr>
<td>10xx</td>
<td>Branches and miscellaneous control</td>
</tr>
<tr>
<td>10x0</td>
<td>Data-processing (modified immediate)</td>
</tr>
<tr>
<td>10x1 xxxx0</td>
<td>Data-processing (plain binary immediate)</td>
</tr>
<tr>
<td>10x1 xxxx1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1100 1xxx0</td>
<td>Advanced SIMD element or structure load/store</td>
</tr>
<tr>
<td>1100 != 1xxx0</td>
<td>Load/store single</td>
</tr>
<tr>
<td>1101 0xxxx</td>
<td>Data-processing (register)</td>
</tr>
<tr>
<td>1101 10xxx</td>
<td>Multiply, multiply accumulate, and absolute difference</td>
</tr>
<tr>
<td>1101 11xxx</td>
<td>Long multiply and divide</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x 0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td><strong>Advanced SIMD data-processing</strong></td>
</tr>
<tr>
<td>0 0x 1x</td>
<td><strong>Advanced SIMD and System register load/store and 64-bit move</strong></td>
</tr>
<tr>
<td>0 10 1x 1</td>
<td><strong>Advanced SIMD and System register 32-bit move</strong></td>
</tr>
<tr>
<td>0 10 10 0</td>
<td><strong>Floating-point data-processing</strong></td>
</tr>
<tr>
<td>0 10 11 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 != 11 1x</td>
<td><strong>Additional Advanced SIMD and floating-point instructions</strong></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>Advanced SIMD three registers of the same length</strong></td>
</tr>
<tr>
<td>1 0</td>
<td><strong>Advanced SIMD two registers, or three registers of different lengths</strong></td>
</tr>
<tr>
<td>1 1</td>
<td><strong>Advanced SIMD shifts and immediate generation</strong></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td><strong>Advanced SIMD three registers of the same length</strong></td>
</tr>
<tr>
<td>1</td>
<td><strong>Advanced SIMD two registers, or three registers of different lengths</strong></td>
</tr>
<tr>
<td>1 1</td>
<td><strong>Advanced SIMD shifts and immediate generation</strong></td>
</tr>
</tbody>
</table>
### Top-level encodings for T32

<table>
<thead>
<tr>
<th>U size</th>
<th>opc</th>
<th>Q</th>
<th>o1</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0x</td>
<td>1100</td>
<td>1</td>
<td></td>
<td>VFMA</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>1101</td>
<td>0</td>
<td></td>
<td>VADD (floating-point)</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>1101</td>
<td>1</td>
<td></td>
<td>VMLA (floating-point)</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>1110</td>
<td>0</td>
<td></td>
<td>VCEQ (register) — T2</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>1111</td>
<td>0</td>
<td></td>
<td>VMAX (floating-point)</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>1111</td>
<td>1</td>
<td></td>
<td>VRECPS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0</td>
<td></td>
<td>VHADD</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>1111</td>
<td>1</td>
<td></td>
<td>VADD (register)</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>0000</td>
<td>1</td>
<td></td>
<td>VQADD</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>0001</td>
<td>0</td>
<td></td>
<td>VRHADD</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>1100</td>
<td>0</td>
<td></td>
<td>SHA1C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>0</td>
<td></td>
<td>VHSUB</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>0001</td>
<td>1</td>
<td></td>
<td>VBIC (register)</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1100</td>
<td>1</td>
<td></td>
<td>VFMS</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1101</td>
<td>0</td>
<td></td>
<td>VSUB (floating-point)</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1101</td>
<td>1</td>
<td></td>
<td>VMLS (floating-point)</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1110</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1111</td>
<td>0</td>
<td></td>
<td>VMIN (floating-point)</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1111</td>
<td>1</td>
<td></td>
<td>VRSORTS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td>0</td>
<td></td>
<td>VSHL (register)</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>1100</td>
<td>0</td>
<td></td>
<td>SHA1P</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1100</td>
<td>1</td>
<td></td>
<td>VFMS</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1101</td>
<td>0</td>
<td></td>
<td>VSUB (floating-point)</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1101</td>
<td>1</td>
<td></td>
<td>VMLS (floating-point)</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1110</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1111</td>
<td>0</td>
<td></td>
<td>VMIN (floating-point)</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1111</td>
<td>1</td>
<td></td>
<td>VRSORTS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td>0</td>
<td></td>
<td>VSHL (register)</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>1100</td>
<td>0</td>
<td></td>
<td>VADD (integer)</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>0001</td>
<td>1</td>
<td></td>
<td>VORR (register)</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>1000</td>
<td>0</td>
<td></td>
<td>VTST</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td>1</td>
<td></td>
<td>VOSHL (register)</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>1001</td>
<td>0</td>
<td></td>
<td>VMLA (integer)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td>0</td>
<td></td>
<td>VRSHL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td>1</td>
<td></td>
<td>VORSRL</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>1011</td>
<td>0</td>
<td></td>
<td>VQDMULH</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>1100</td>
<td>0</td>
<td></td>
<td>SHA1M</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>1101</td>
<td>1</td>
<td></td>
<td>VPADD (integer)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110</td>
<td>0</td>
<td></td>
<td>VMAX (integer)</td>
<td></td>
</tr>
<tr>
<td>0 11</td>
<td>0001</td>
<td>1</td>
<td></td>
<td>VORN (register)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110</td>
<td>1</td>
<td></td>
<td>VMIN (integer)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111</td>
<td>0</td>
<td></td>
<td>VABD (integer)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111</td>
<td>1</td>
<td></td>
<td>VABA</td>
<td></td>
</tr>
<tr>
<td>0 11</td>
<td>1100</td>
<td>0</td>
<td></td>
<td>SHA1SU0</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>1101</td>
<td>0</td>
<td></td>
<td>VPADD (floating-point)</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>1101</td>
<td>1</td>
<td></td>
<td>VMUL (floating-point)</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>1110</td>
<td>0</td>
<td></td>
<td>VCEQ (register) — T2</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>1110</td>
<td>1</td>
<td></td>
<td>VACGE</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>1111</td>
<td>0</td>
<td></td>
<td>VPMAX (floating-point)</td>
<td></td>
</tr>
<tr>
<td>1 0x</td>
<td>1111</td>
<td>1</td>
<td></td>
<td>VMAXNM</td>
<td></td>
</tr>
<tr>
<td>1 00</td>
<td>0001</td>
<td>1</td>
<td></td>
<td>VEOR</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD two registers, or three registers of different lengths

These instructions are under Advanced SIMD data-processing.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>size</th>
<th>opc</th>
<th>Q</th>
<th>o1</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>00</td>
<td>1100</td>
<td>0</td>
<td>0</td>
<td>SHA256H</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>01</td>
<td>0001</td>
<td>1</td>
<td>0</td>
<td>VPMAX (integer)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>01</td>
<td>1100</td>
<td>0</td>
<td>1</td>
<td>VPMMIN (integer)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1x</td>
<td>1101</td>
<td>0</td>
<td>0</td>
<td>VABD (floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1x</td>
<td>1110</td>
<td>0</td>
<td>1</td>
<td>VACGT</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1x</td>
<td>1110</td>
<td>0</td>
<td>0</td>
<td>VPMMIN (floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1x</td>
<td>1111</td>
<td>1</td>
<td>0</td>
<td>VMINNM</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>VSUB (integer)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VBIT</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1001</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>VMLS (integer)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1011</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>VORDMULH</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1011</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SHA256SU1</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VORDMLAH</td>
<td>Armv8.1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VORDMLSH</td>
<td>Armv8.1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Advanced SIMD two registers misc

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>size</th>
<th>opc1</th>
<th>Q</th>
<th>Vd</th>
<th>opc2</th>
<th>Q</th>
<th>M</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000</td>
<td>00</td>
<td>VREV64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0001</td>
<td>01</td>
<td>VREV32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>Decode fields</td>
<td>Q</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>--------------</td>
<td>---</td>
<td>---------------------</td>
<td>----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00 0010</td>
<td></td>
<td>VREV16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00 0011</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01 010x</td>
<td></td>
<td>VPADDL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01 0110</td>
<td>0</td>
<td>AESE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01 0110</td>
<td>1</td>
<td>AESD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01 0111</td>
<td>0</td>
<td>AESMC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01 0111</td>
<td>1</td>
<td>AESIMC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10 000</td>
<td></td>
<td>VCLS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10 0000</td>
<td></td>
<td>VSWP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10 0001</td>
<td></td>
<td>VCLZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10 0100</td>
<td></td>
<td>VCNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10 0110</td>
<td></td>
<td>VMVN (register)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10 1100</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>11 010</td>
<td></td>
<td>VPADAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>11 1100</td>
<td></td>
<td>VOABS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>11 111</td>
<td></td>
<td>VONEG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x0 000</td>
<td></td>
<td>VCGT (immediate #0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x0 001</td>
<td></td>
<td>VCGE (immediate #0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x0 010</td>
<td></td>
<td>VCEO (immediate #0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x0 011</td>
<td></td>
<td>VCLE (immediate #0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x1 000</td>
<td></td>
<td>VCLT (immediate #0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x1 100</td>
<td></td>
<td>VABS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x1 110</td>
<td></td>
<td>VNEG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01 0101</td>
<td>1</td>
<td>SHA1H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>10 1100</td>
<td>1</td>
<td>VCVT (from single-precision to BFloat16, Advanced SIMD)</td>
<td>Armv8.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00 001</td>
<td></td>
<td>VTRN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00 010</td>
<td></td>
<td>VUZP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00 011</td>
<td></td>
<td>VZIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01 0100</td>
<td>0</td>
<td>VMOVN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01 0100</td>
<td>1</td>
<td>VQMOVN, VQMOVUN — VQMOVN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01 0101</td>
<td></td>
<td>VQMOVN, VQMOVUN — VQMOVN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01 1100</td>
<td>0</td>
<td>VSHLL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01 1110</td>
<td>0</td>
<td>SHA1SU1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01 1110</td>
<td>1</td>
<td>SHA256SU0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10 000</td>
<td></td>
<td>VRINTN (Advanced SIMD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10 001</td>
<td></td>
<td>VRINTX (Advanced SIMD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10 100</td>
<td></td>
<td>VRINTA (Advanced SIMD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10 1100</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11 000</td>
<td></td>
<td>VCVT (between half-precision and single-precision, Advanced SIMD) — single-precision to half-precision</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11 110</td>
<td></td>
<td>VRINTM (Advanced SIMD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11 1100</td>
<td>0</td>
<td>VCVT (between half-precision and single-precision, Advanced SIMD) — half-precision to single-precision</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11 1101</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11 111</td>
<td></td>
<td>VRINTP (Advanced SIMD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00 000x</td>
<td></td>
<td>VCVTA (Advanced SIMD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00 01x</td>
<td></td>
<td>VCVTN (Advanced SIMD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Top-level encodings for T32

<table>
<thead>
<tr>
<th>size</th>
<th>Decode fields</th>
<th>opc1</th>
<th>opc2</th>
<th>Q</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>010x</td>
<td></td>
<td></td>
<td></td>
<td>VCVTP (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011x</td>
<td></td>
<td></td>
<td></td>
<td>VCVTM (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>10x0</td>
<td></td>
<td></td>
<td></td>
<td>VRECPE</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>10x1</td>
<td></td>
<td></td>
<td></td>
<td>VRSORTE</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>10 1100 1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11xx</td>
<td></td>
<td></td>
<td></td>
<td>VCVT (between floating-point and integer, Advanced SIMD)</td>
<td>-</td>
</tr>
</tbody>
</table>

**Advanced SIMD duplicate (scalar)**

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>VDUP (scalar)</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**Advanced SIMD three registers of different lengths**

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>VADDL</td>
</tr>
<tr>
<td>0001</td>
<td>VADDW</td>
</tr>
<tr>
<td>0010</td>
<td>VSUBL</td>
</tr>
<tr>
<td>0 0100</td>
<td>VADDHN</td>
</tr>
<tr>
<td>0011</td>
<td>VSUBW</td>
</tr>
<tr>
<td>0 0110</td>
<td>VSUBHN</td>
</tr>
<tr>
<td>0 1001</td>
<td>VQDMLAL</td>
</tr>
<tr>
<td>0101</td>
<td>VABAL</td>
</tr>
<tr>
<td>0 1011</td>
<td>VQDMLSL</td>
</tr>
<tr>
<td>0 1101</td>
<td>VQDMULL</td>
</tr>
<tr>
<td>0111</td>
<td>VABDL (integer)</td>
</tr>
<tr>
<td>1000</td>
<td>VMLAL (integer)</td>
</tr>
<tr>
<td>1010</td>
<td>VMLSL (integer)</td>
</tr>
<tr>
<td>1 0100</td>
<td>VRADDHN</td>
</tr>
<tr>
<td>1 0110</td>
<td>VRSUBHN</td>
</tr>
<tr>
<td>11x0</td>
<td>VMULL (integer and polynomial)</td>
</tr>
<tr>
<td>1 1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1011</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: size != 11 && size != 11
Advanced SIMD two registers and a scalar

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

The following constraints also apply to this encoding: size != 11 && size != 11

Advanced SIMD shifts and immediate generation

These instructions are under Advanced SIMD data-processing.

Advanced SIMD one register and modified immediate

These instructions are under Advanced SIMD shifts and immediate generation.
### Advanced SIMD two registers and shift amount

These instructions are under Advanced SIMD shifts and immediate generation.

<table>
<thead>
<tr>
<th>U</th>
<th>imm3H:L</th>
<th>imm3L</th>
<th>opc</th>
<th>Q</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 0000</td>
<td>0000</td>
<td>0000</td>
<td>VSHR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>! = 0000</td>
<td>0000</td>
<td>0001</td>
<td>VSRA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>! = 0000</td>
<td>0000</td>
<td>0000</td>
<td>0110</td>
<td>0</td>
<td>VMOVL</td>
</tr>
<tr>
<td>! = 0000</td>
<td>0000</td>
<td>0010</td>
<td>VRSRA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>! = 0000</td>
<td>0111</td>
<td>0000</td>
<td>0</td>
<td>VOSHL, VOSHLU (immediate) — VOSHL</td>
<td></td>
</tr>
<tr>
<td>! = 0000</td>
<td>0000</td>
<td>0100</td>
<td>0</td>
<td>VOSHRN, VOSHRUN — VOSHRN</td>
<td></td>
</tr>
<tr>
<td>! = 0000</td>
<td>0000</td>
<td>1001</td>
<td>1</td>
<td>VQRSHRN, VQRSHRUN — VQRSHRN</td>
<td></td>
</tr>
<tr>
<td>! = 0000</td>
<td>0000</td>
<td>1010</td>
<td>0</td>
<td>VSHLL</td>
<td></td>
</tr>
<tr>
<td>! = 0000</td>
<td>11xx</td>
<td>0000</td>
<td>0</td>
<td>VCVT (between floating-point and fixed-point, Advanced SIMD)</td>
<td></td>
</tr>
<tr>
<td>0 ! = 0000</td>
<td>0000</td>
<td>0101</td>
<td>VSHL (immediate)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ! = 0000</td>
<td>1000</td>
<td>0</td>
<td>VSHRN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ! = 0000</td>
<td>1000</td>
<td>1</td>
<td>VQRSHRN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ! = 0000</td>
<td>0000</td>
<td>0100</td>
<td>VSRI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ! = 0000</td>
<td>0100</td>
<td>0101</td>
<td>VSLI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ! = 0000</td>
<td>0110</td>
<td>0</td>
<td>VOSHL, VOSHLU (immediate) — VOSHL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ! = 0000</td>
<td>1000</td>
<td>0</td>
<td>VQRSHRN, VQRSHRUN — VQRSHRUN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ! = 0000</td>
<td>1000</td>
<td>1</td>
<td>VQRSHRN, VQRSHRUN — VQRSHRUN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Advanced SIMD and System register load/store and 64-bit move

These instructions are under System register access, Advanced SIMD, and floating-point.

<table>
<thead>
<tr>
<th>U</th>
<th>op0</th>
<th>op1</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110110</td>
<td>0</td>
<td>1</td>
<td>Vmove (immediate) — T1</td>
</tr>
<tr>
<td>10x0</td>
<td>0</td>
<td>VMOV (immediate) — T3</td>
<td></td>
</tr>
<tr>
<td>10x0</td>
<td>1</td>
<td>VMVN (immediate) — T2</td>
<td></td>
</tr>
<tr>
<td>10x1</td>
<td>0</td>
<td>VORR (immediate) — T2</td>
<td></td>
</tr>
<tr>
<td>10x1</td>
<td>1</td>
<td>VBIC (immediate) — T2</td>
<td></td>
</tr>
<tr>
<td>0x1</td>
<td>0</td>
<td>VMOV (immediate) — T4</td>
<td></td>
</tr>
<tr>
<td>0x1</td>
<td>1</td>
<td>VMVN (immediate) — T3</td>
<td></td>
</tr>
<tr>
<td>0x1</td>
<td>0</td>
<td>VMOV (immediate) — T5</td>
<td></td>
</tr>
<tr>
<td>0x1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
Advanced SIMD and floating-point 64-bit move

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x0 0x</td>
<td>Advanced SIMD and floating-point 64-bit move</td>
</tr>
<tr>
<td>00x0 11</td>
<td>System register 64-bit move</td>
</tr>
<tr>
<td>!= 00x0 0x</td>
<td>Advanced SIMD and floating-point load/store</td>
</tr>
<tr>
<td>!= 00x0 11</td>
<td>System register Load/Store</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

System register 64-bit move

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0 UNALLOCATED</td>
</tr>
<tr>
<td>1 0x 00 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0 10 00 1</td>
<td>VMOV (between two general-purpose registers and two single-precision registers) — from general-purpose registers</td>
</tr>
<tr>
<td>1 0 11 00 1</td>
<td>VMOV (between two general-purpose registers and a doubleword floating-point register) — from general-purpose registers</td>
</tr>
<tr>
<td>1 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1 10 00 1</td>
<td>VMOV (between two general-purpose registers and two single-precision registers) — to general-purpose registers</td>
</tr>
<tr>
<td>1 1 11 00 1</td>
<td>VMOV (between two general-purpose registers and a doubleword floating-point register) — to general-purpose registers</td>
</tr>
</tbody>
</table>

Advanced SIMD and floating-point load/store

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>MCRR</td>
</tr>
<tr>
<td>1 1</td>
<td>MBRC</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: P:U:D:W != 00x0
## System register Load/Store

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

The following constraints also apply to this encoding: P:U:D:W != 00x0

## Advanced SIMD and System register 32-bit move

These instructions are under System register access, Advanced SIMD, and floating-point.
### Floating-point move special register

These instructions are under [Advanced SIMD and System register 32-bit move](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>VMSR</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>VMRS</td>
</tr>
</tbody>
</table>

### Advanced SIMD 8/16/32-bit element move/duplicate

These instructions are under [Advanced SIMD and System register 32-bit move](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>opc1</th>
<th>L</th>
<th>opc2</th>
<th>Vn</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x</td>
<td></td>
<td></td>
<td></td>
<td>VMOV (general-purpose register to scalar)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>0x</td>
<td></td>
<td>VMOV (scalar to general-purpose register)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
<td>VDUP (general-purpose register)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### System register 32-bit move

These instructions are under [Advanced SIMD and System register 32-bit move](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>MCR</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>MRC</td>
</tr>
</tbody>
</table>
Floating-point data-processing

These instructions are under System register access, Advanced SIMD, and floating-point.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>1</td>
<td>Floating-point data-processing (two registers)</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0</td>
<td>Floating-point move immediate</td>
</tr>
<tr>
<td>!= 1x11</td>
<td></td>
<td>Floating-point data-processing (three registers)</td>
</tr>
</tbody>
</table>

Floating-point data-processing (two registers)

These instructions are under Floating-point data-processing.

<table>
<thead>
<tr>
<th>o1</th>
<th>Decode fields</th>
<th>size</th>
<th>o3</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>01</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>1</td>
<td>VABS</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1</td>
<td>0</td>
<td>VMOV (register) — single-precision scalar</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1</td>
<td>1</td>
<td>VMOV (register) — double-precision scalar</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1</td>
<td>0</td>
<td>VNEG</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1</td>
<td>1</td>
<td>VSORT</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>0</td>
<td>VCVTB — half-precision to double-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>1</td>
<td>VCVTT — half-precision to double-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>0</td>
<td>VCVTB (BFloat16)</td>
<td>Armv8.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 0</td>
<td>1</td>
<td>VCVTB (BFloat16)</td>
<td>Armv8.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 0</td>
<td>0</td>
<td>VCVTB — single-precision to half-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 0</td>
<td>1</td>
<td>VCVT — single-precision to half-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 1</td>
<td>0</td>
<td>VCVTB — double-precision to half-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 1</td>
<td>1</td>
<td>VCVT — double-precision to half-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>0</td>
<td>VCMPE — T1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 1</td>
<td>0</td>
<td>VCMPE — T1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 1</td>
<td>1</td>
<td>VCMPE — T2</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 0</td>
<td>0</td>
<td>VRINTR</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 1</td>
<td>0</td>
<td>VINTZ (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 1</td>
<td>0</td>
<td>VINTZ (floating-point)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 0</td>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 0</td>
<td>1</td>
<td>VCVT (between double-precision and single-precision) — single-precision to double-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 1</td>
<td>0</td>
<td>VCVT (between double-precision and single-precision) — double-precision to single-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0 0</td>
<td>1</td>
<td>VCVT (integer to floating-point, floating-point)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0 1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Top-level encodings for T32

<table>
<thead>
<tr>
<th>o1</th>
<th>Decode fields</th>
<th>o2</th>
<th>size</th>
<th>o3</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
<td>11</td>
<td>1</td>
<td>VVCVT</td>
<td></td>
<td>Armv8.3</td>
</tr>
<tr>
<td>1</td>
<td>01x</td>
<td></td>
<td></td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>VCVTR</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>VCVTR</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11x</td>
<td></td>
<td></td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Floating-point move immediate

These instructions are under Floating-point data-processing.

<table>
<thead>
<tr>
<th>11</th>
<th>11</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>D</th>
<th>1</th>
<th>imm4H</th>
<th>Vd</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>imm4L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>VMOV (immediate) — half-precision scalar</td>
<td>Armv8.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>VMOV (immediate) — single-precision scalar</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>VMOV (immediate) — double-precision scalar</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Floating-point data-processing (three registers)

These instructions are under Floating-point data-processing.

<table>
<thead>
<tr>
<th>11</th>
<th>11</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>D</th>
<th>0</th>
<th>1</th>
<th>Vn</th>
<th>Vd</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>0</th>
<th>0</th>
<th>M</th>
<th>0</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>VMLA (floating-point)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>VMLS (floating-point)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>VNMLS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>VNMLA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>VMUL (floating-point)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>VNML</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>VADD (floating-point)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>VSUB (floating-point)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>VDIV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>VFMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>VFMA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>VFMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional Advanced SIMD and floating-point instructions

These instructions are under System register access, Advanced SIMD, and floating-point.
The following constraints also apply to this encoding: \( \text{op0}<2:1> \neq 11 \)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Decode fields</th>
<th>op4</th>
<th>op5</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>0x</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>Advanced SIMD three registers of the same length extension</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>!= 00</td>
<td>0</td>
<td>0</td>
<td>VSELEQ, VSELGE, VSELGT, VSELVS</td>
</tr>
<tr>
<td>101</td>
<td>00xxx</td>
<td>!= 00</td>
<td>0</td>
<td>0</td>
<td>Floating-point minNum/maxNum</td>
</tr>
<tr>
<td>101</td>
<td>110000</td>
<td>!= 00</td>
<td>1</td>
<td>0</td>
<td>Floating-point extraction and insertion</td>
</tr>
<tr>
<td>101</td>
<td>111xxx</td>
<td>!= 00</td>
<td>1</td>
<td>0</td>
<td>Floating-point directed convert to integer</td>
</tr>
<tr>
<td>10x</td>
<td>0</td>
<td>00</td>
<td></td>
<td></td>
<td>Advanced SIMD and floating-point multiply with accumulate</td>
</tr>
<tr>
<td>10x</td>
<td>1</td>
<td>0x</td>
<td></td>
<td></td>
<td>Advanced SIMD and floating-point dot product</td>
</tr>
</tbody>
</table>

**Advanced SIMD three registers of the same length extension**

These instructions are under [Additional Advanced SIMD and floating-point instructions](#).

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>Decode fields</th>
<th>op3</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>0x</td>
<td>0 0 0 0 0 0 0</td>
<td>VCAD — 64-bit SIMD vector</td>
<td>Armv8.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>0x</td>
<td>0 0 1 1 1 1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>0x</td>
<td>0 0 1 0 0 0 0</td>
<td>VCAD — 128-bit SIMD vector</td>
<td>Armv8.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>0x</td>
<td>0 0 1 1 1 1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0x</td>
<td>0 0 0 0 0 0 0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0x</td>
<td>0 0 0 0 1 1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>0 0 0 0 0 0 0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>0 0 1 1 1 1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>0 0 1 1 1 1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>0 0 0 0 0 0 0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>0 0 0 0 1 1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0 0 0 0 0 0 0</td>
<td>VFMAL (vector)</td>
<td>Armv8.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0 0 0 0 1 1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0 0 0 0 1 1 1</td>
<td>VSMMLA</td>
<td>Armv8.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0 0 0 0 1 1 1</td>
<td>VUMMLA</td>
<td>Armv8.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0 0 0 0 1 1 1</td>
<td>VSDOT (vector) — 64-bit SIMD vector</td>
<td>Armv8.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0 0 0 0 1 1 1</td>
<td>VUDOT (vector) — 64-bit SIMD vector</td>
<td>Armv8.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0 0 0 0 1 1 1</td>
<td>VSDOT (vector) — 128-bit SIMD vector</td>
<td>Armv8.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Floating-point minNum/maxNum

These instructions are under the [Additional Advanced SIMD and floating-point instructions](#).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>!= 00</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: size != 00 && size != 00

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1</td>
<td>VMAXNM</td>
</tr>
<tr>
<td>op2</td>
<td>VMINNM</td>
</tr>
</tbody>
</table>

### Floating-point extraction and insertion

These instructions are under the [Additional Advanced SIMD and floating-point instructions](#).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Vd</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: size != 00 && size != 00
Floating-point directed convert to integer

These instructions are under Additional Advanced SIMD and floating-point instructions.

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 |o1| RM | Vd | 1 0 | != 00 | op | 1 | M | 0 | Vm
  size
```

The following constraints also apply to this encoding: size != 00 && size != 00

<table>
<thead>
<tr>
<th>o1</th>
<th>Decode fields</th>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>!= 00</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>VRINTA (floating-point)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td>VRINTN (floating-point)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>VRINTP (floating-point)</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0</td>
<td>VRINTM (floating-point)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td>VCVTA (floating-point)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0</td>
<td>VCVTN (floating-point)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td>VCVTP (floating-point)</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td>VCVTM (floating-point)</td>
</tr>
</tbody>
</table>

Advanced SIMD and floating-point multiply with accumulate

These instructions are under Additional Advanced SIMD and floating-point instructions.

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 0 |op1| D | op2 | Vn | Vd | 1 0 0 0 | N | Q | M | U | Vm
```

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>Q</th>
<th>U</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>VCMILA (by element) — 128-bit SIMD vector of half-precision floating-point</td>
<td>Armv8.3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td>VFMAI (by scalar)</td>
<td>Armv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1</td>
<td>VFMSL (by scalar)</td>
<td>Armv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1</td>
<td>VFMB, VFMAT (BFLOAT16, by scalar)</td>
<td>Armv8.6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VCMILA (by element) — 64-bit SIMD vector of single-precision floating-point</td>
<td>Armv8.3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VCMILA (by element) — 128-bit SIMD vector of single-precision floating-point</td>
<td>Armv8.3</td>
<td></td>
</tr>
</tbody>
</table>

Advanced SIMD and floating-point dot product

These instructions are under Additional Advanced SIMD and floating-point instructions.
### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VDOT (by element) — 128-bit SIMD vector</td>
<td>Armv8.6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>VSDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>VUDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>VSDOT (by element) — 128-bit SIMD vector</td>
<td>Armv8.2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>VUSDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>VSUDOT (by element) — 64-bit SIMD vector</td>
<td>Armv8.6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>VSUDOT (by element) — 128-bit SIMD vector</td>
<td>Armv8.6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>VSUDOT (by element) — 128-bit SIMD vector</td>
<td>Armv8.6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Load/store multiple

These instructions are under **32-bit**.

<table>
<thead>
<tr>
<th>opc</th>
<th>W</th>
<th>L</th>
<th>Rn</th>
<th>P</th>
<th>M</th>
<th>register_list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

These instructions are under **32-bit**.

The following constraints also apply to this encoding: op0<1> == 1

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>D</td>
<td>SRS, SRSDB, SRSIA, SRSIB — T1</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
<td>SRS, SRSDB, SRSIA, SRSIB — T1</td>
</tr>
<tr>
<td>Vd</td>
<td>Vd</td>
<td>RFE, RFEDB, RFEIA, RFEIB — T1</td>
</tr>
<tr>
<td>1</td>
<td>L</td>
<td>STM, STMIA, STMEA</td>
</tr>
<tr>
<td>1</td>
<td>L</td>
<td>LDM, LDMIA, LDMFD</td>
</tr>
<tr>
<td>1</td>
<td>L</td>
<td>STMDB, STMFD</td>
</tr>
<tr>
<td>1</td>
<td>L</td>
<td>LDMDB, LDMEA</td>
</tr>
<tr>
<td>0</td>
<td>SRS, SRSDB, SRSIA, SRSIB — T2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>RFE, RFEDB, RFEIA, RFEIB — T2</td>
<td></td>
</tr>
</tbody>
</table>
### Load/store exclusive

These instructions are under **Load/store dual, load/store exclusive, load-acquire/store-release, and table branch**.

<table>
<thead>
<tr>
<th>Field</th>
<th>Decode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0</td>
<td>Load/store exclusive</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0110</td>
<td>1</td>
<td>TBB, TBH</td>
</tr>
<tr>
<td>0110</td>
<td>01x</td>
<td>Load/store exclusive byte/half/dual</td>
</tr>
<tr>
<td>0110</td>
<td>1xx</td>
<td>Load-acquire / Store-release</td>
</tr>
<tr>
<td>0x11</td>
<td>!= 1111</td>
<td>Load/store dual (immediate, post-indexed)</td>
</tr>
<tr>
<td>1x10</td>
<td>!= 1111</td>
<td>Load/store dual (immediate)</td>
</tr>
<tr>
<td>1x11</td>
<td>!= 1111</td>
<td>Load/store dual (immediate, pre-indexed)</td>
</tr>
<tr>
<td>!= 0xx0</td>
<td>1111</td>
<td>LDRD (literal)</td>
</tr>
</tbody>
</table>

### Load/store exclusive byte/half/dual

These instructions are under **Load/store dual, load/store exclusive, load-acquire/store-release, and table branch**.

<table>
<thead>
<tr>
<th>Field</th>
<th>Decode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
<td>00</td>
<td>STREXB</td>
</tr>
<tr>
<td>0x11</td>
<td>01</td>
<td>STREXH</td>
</tr>
<tr>
<td>0x11</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x11</td>
<td>11</td>
<td>STREXD</td>
</tr>
<tr>
<td>1x11</td>
<td>00</td>
<td>LDREXB</td>
</tr>
<tr>
<td>1x11</td>
<td>01</td>
<td>LDREXH</td>
</tr>
<tr>
<td>1x11</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x11</td>
<td>11</td>
<td>LDREXD</td>
</tr>
</tbody>
</table>

### Load-acquire / Store-release

These instructions are under **Load/store dual, load/store exclusive, load-acquire/store-release, and table branch**.

<table>
<thead>
<tr>
<th>Field</th>
<th>Decode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>00</td>
<td>STLB</td>
</tr>
<tr>
<td>0010</td>
<td>01</td>
<td>STLH</td>
</tr>
<tr>
<td>0010</td>
<td>10</td>
<td>STL</td>
</tr>
<tr>
<td>0010</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

Top-level encodings for T32
Load/store dual (immediate, post-indexed)

These instructions are under **Load/store dual, load/store exclusive, load-acquire/store-release, and table branch**.

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>op</th>
<th>sz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>STLEXB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>STLEXH</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>STLEX</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>STLEXD</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>LDAB</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>LDAH</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>LDA</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>LDAEXB</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>LDAEXH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>LDAEX</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>LDAEXD</td>
</tr>
</tbody>
</table>

Load/store dual (immediate)

These instructions are under **Load/store dual, load/store exclusive, load-acquire/store-release, and table branch**.

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STRD (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>LDRD (immediate)</td>
</tr>
</tbody>
</table>

Load/store dual (immediate, pre-indexed)

These instructions are under **Load/store dual, load/store exclusive, load-acquire/store-release, and table branch**.

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STRD (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>LDRD (immediate)</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STRD (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>LDRD (immediate)</td>
</tr>
</tbody>
</table>

### Data-processing (shifted register)

These instructions are under 32-bit.

<table>
<thead>
<tr>
<th>op1</th>
<th>S</th>
<th>Rn</th>
<th>imm3:imm2:stype</th>
<th>Rd</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>AND, ANDS (register) — AND, rotate right with extend</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
<td></td>
<td>!= 0000011</td>
<td>1111</td>
<td>AND, ANDS (register) — ANDS, shift or rotate by value</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
<td></td>
<td>!= 0000011</td>
<td>1111</td>
<td>TST (register) — shift or rotate by value</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
<td></td>
<td>0000011</td>
<td>!= 1111</td>
<td>AND, ANDS (register) — ANDS, rotate right with extend</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
<td></td>
<td>0000011</td>
<td>1111</td>
<td>TST (register) — rotate right with extend</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BIC, BICS (register)</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td></td>
<td>!= 1111</td>
<td></td>
<td>ORR, ORRS (register) — ORR</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td></td>
<td>1111</td>
<td></td>
<td>MOV, MOVS (register) — MOV</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td></td>
<td>!= 1111</td>
<td></td>
<td>ORR, ORRS (register) — ORRS</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td></td>
<td>1111</td>
<td></td>
<td>MOV, MOVS (register) — MOVS</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td></td>
<td>!= 1111</td>
<td></td>
<td>ORN, ORNS (register) — not flag setting</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td></td>
<td>1111</td>
<td></td>
<td>MVN, MVNS (register) — MVN</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td></td>
<td>!= 1111</td>
<td></td>
<td>ORN, ORNS (register) — flag setting</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td></td>
<td>1111</td>
<td></td>
<td>MVN, MVNS (register) — MVNS</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td></td>
<td>!= 0000011</td>
<td>1111</td>
<td>FOR, EORS (register) — FOR, rotate right with extend</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td></td>
<td>!= 0000011</td>
<td>1111</td>
<td>FOR, EORS (register) — EORS, shift or rotate by value</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td></td>
<td>!= 0000011</td>
<td>1111</td>
<td>TEQ (register) — shift or rotate by value</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td></td>
<td>0000011</td>
<td>!= 1111</td>
<td>FOR, EORS (register) — EORS, rotate right with extend</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td></td>
<td>0000011</td>
<td>1111</td>
<td>TEQ (register) — rotate right with extend</td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td></td>
<td>xxxxx00</td>
<td></td>
<td>PKHBT, PKHTB — PKHBT</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td></td>
<td>xxxxx01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td></td>
<td>xxxxx10</td>
<td></td>
<td>PKHBT, PKHTB — PKHTB</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td></td>
<td>xxxxx11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td></td>
<td>!= 1101</td>
<td></td>
<td>ADD, ADDS (register) — ADD</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td></td>
<td>1101</td>
<td></td>
<td>ADD, ADDS (SP plus register) — ADD</td>
</tr>
</tbody>
</table>
Top-level encodings for T32

### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>S</th>
<th>Rn</th>
<th>imm3:imm2:stype</th>
<th>Rd</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>!1101</td>
<td>!= 1111</td>
<td>ADD, ADDS (register) — ADDS</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1101</td>
<td>!= 1111</td>
<td>ADD, ADDS (SP plus register) — ADDS</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1111</td>
<td>CMN (register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td>ADC, ADCS (register)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td>SBC, SBCS (register)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>!1101</td>
<td>SUB, SUBS (register) — SUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>1101</td>
<td>SUB, SUBS (SP minus register) — SUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>!1111</td>
<td>SUB, SUBS (register) — SUBS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1101</td>
<td>SUB, SUBS (SP minus register) — SUBS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1111</td>
<td>CMP (register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td>RSB, RSBS (register)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Branches and miscellaneous control

These instructions are under **32-bit**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>op5</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1110</td>
<td>0x 0x0</td>
<td>0</td>
<td>MSR (register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>0x 0x0</td>
<td>000</td>
<td>MSR (Banked register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>10 0x0</td>
<td>!= 000</td>
<td>Change processor state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>11 0x0</td>
<td>0</td>
<td>Miscellaneous system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>00 0x0</td>
<td>0</td>
<td>BXI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>01 0x0</td>
<td>0</td>
<td>Exception return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>1x 0x0</td>
<td>0</td>
<td>MRS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>1x 0x0</td>
<td>1</td>
<td>MRS (Banked register)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>00 000</td>
<td>0</td>
<td>DCPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>00 010</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>01 0x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>1x 0x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>0x 0x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>1x 0x0</td>
<td>0</td>
<td>Exception generation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!= 111x</td>
<td>0x0</td>
<td>0x1</td>
<td>1x0</td>
<td>B — T3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BL, BLX (immediate) — T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BL, BLX (immediate) — T1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hints

These instructions are under **Branches and miscellaneous control**.

<table>
<thead>
<tr>
<th>Decode fields hint</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>NOP</td>
<td>-</td>
</tr>
<tr>
<td>0000 0001</td>
<td>YIELD</td>
<td>-</td>
</tr>
<tr>
<td>0000 0010</td>
<td>WFE</td>
<td>-</td>
</tr>
<tr>
<td>0000 0011</td>
<td>WFI</td>
<td>-</td>
</tr>
<tr>
<td>0000 0100</td>
<td>SEV</td>
<td>-</td>
</tr>
<tr>
<td>0000 0101</td>
<td>SEVL</td>
<td>-</td>
</tr>
<tr>
<td>0000 011x</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>0000 1xxx</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>0001 0000</td>
<td>ESB</td>
<td>Armv8.2</td>
</tr>
<tr>
<td>0001 0001</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>0001 0010</td>
<td>TSB.CSYNC</td>
<td>Armv8.4</td>
</tr>
<tr>
<td>0001 0011</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>0001 0100</td>
<td>CSDB</td>
<td>-</td>
</tr>
<tr>
<td>0001 0101</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>0001 011x</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>0001 1xxx</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>01xx</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>10xx</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>110x</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>1110</td>
<td>Reserved hint, behaves as NOP</td>
<td>-</td>
</tr>
<tr>
<td>1111</td>
<td>DBG</td>
<td>-</td>
</tr>
</tbody>
</table>

Change processor state

These instructions are under **Branches and miscellaneous control**.

<table>
<thead>
<tr>
<th>Decode fields imod</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>CPS, CPSID, CPSIE — change mode</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>CPS, CPSID, CPSIE — interrupt enable and change mode</td>
</tr>
<tr>
<td>11</td>
<td>CPS, CPSID, CPSIE — interrupt disable and change mode</td>
</tr>
</tbody>
</table>

Miscellaneous system

These instructions are under **Branches and miscellaneous control**.
### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>option</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>CLREX</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>!= 0x00 DSB</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>0000 SSBB</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>0100 PSSBB</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>DMB</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>ISB</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>SB</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

### Exception return

These instructions are under **Branches and miscellaneous control**.

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 0 1 1 1 0 1 Rn 1 0 [0][0][1][1][1][1] imm8
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>! = 00000000 SUB_SUBS (immediate)</td>
</tr>
<tr>
<td>imm8</td>
<td>ERET</td>
</tr>
</tbody>
</table>

### DCPS

These instructions are under **Branches and miscellaneous control**.

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 1 1 1 1 0 0 0 imm4 1 0 0 0 imm10 opt
```

<table>
<thead>
<tr>
<th>imm4</th>
<th>Decode fields</th>
<th>opt</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 1111</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1111</td>
<td>! = 0000000000</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1111</td>
<td>0000000000</td>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1111</td>
<td>0000000000</td>
<td>01</td>
<td>DCPS1</td>
</tr>
<tr>
<td>1111</td>
<td>0000000000</td>
<td>10</td>
<td>DCPS2</td>
</tr>
<tr>
<td>1111</td>
<td>0000000000</td>
<td>11</td>
<td>DCPS3</td>
</tr>
</tbody>
</table>

### Exception generation

These instructions are under **Branches and miscellaneous control**.

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0 1 1 1 1 1 1 o1 imm4 1 0 o2 0 imm12
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>HVC</td>
</tr>
<tr>
<td>o2</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>SMC</td>
</tr>
<tr>
<td>1</td>
<td>UDF</td>
</tr>
</tbody>
</table>
Data-processing (modified immediate)

These instructions are under 32-bit.

<table>
<thead>
<tr>
<th>op1</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td></td>
<td></td>
<td>AND, ANDS (immediate) — AND</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>AND, ANDS (immediate) — ANDS</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
<td></td>
<td></td>
<td>TST (immediate)</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td></td>
<td>1111</td>
<td>BIC, BICS (immediate)</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td></td>
<td>1111</td>
<td>ORR, ORRS (immediate) — ORR</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>MOV, MOVS (immediate) — MOV</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>ORR, ORRS (immediate) — ORRS</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td></td>
<td>1111</td>
<td>MOV, MOVS (immediate) — MOVS</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>ORN, ORNS (immediate) — not flag setting</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>MVN, MVNS (immediate) — MVN</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td></td>
<td>1111</td>
<td>EOR, EORS (immediate) — EOR</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>EOR, EORS (immediate) — EORS</td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>TEO (immediate)</td>
</tr>
<tr>
<td>011x</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td></td>
<td>1101</td>
<td>ADD, ADDS (immediate) — ADD</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td></td>
<td>1101</td>
<td>ADD, ADDS (SP plus immediate) — ADD</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>ADD, ADDS (immediate) — ADDS</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>ADD, ADDS (SP plus immediate) — ADDS</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>CMN (immediate)</td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td>ADC, ADCS (immediate)</td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td>SBC, SBCS (immediate)</td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td></td>
<td>1101</td>
<td>SUB, SUBS (immediate) — SUB</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td></td>
<td>1101</td>
<td>SUB, SUBS (SP minus immediate) — SUB</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>SUB, SUBS (immediate) — SUBS</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>SUB, SUBS (SP minus immediate) — SUBS</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>CMP (immediate)</td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td>RSB, RSBS (immediate)</td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Data-processing (plain binary immediate)

These instructions are under 32-bit.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x</td>
<td>Data-processing (simple immediate)</td>
</tr>
</tbody>
</table>
Data-processing (simple immediate)

These instructions are under **Data-processing (plain binary immediate)**.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>i</td>
<td>1</td>
<td>0</td>
<td>o1</td>
<td>0</td>
<td>o2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1 o2 Rn</td>
<td></td>
</tr>
<tr>
<td>0 0 ! = 11x1</td>
<td>ADD, ADDS (immediate)</td>
</tr>
<tr>
<td>0 0 1101</td>
<td>ADD, ADDS (SP plus immediate)</td>
</tr>
<tr>
<td>0 0 1111</td>
<td>ADR — T3</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1 ! = 11x1</td>
<td>SUB, SUBS (immediate)</td>
</tr>
<tr>
<td>1 1 1101</td>
<td>SUB, SUBS (SP minus immediate)</td>
</tr>
<tr>
<td>1 1 1111</td>
<td>ADR — T2</td>
</tr>
</tbody>
</table>

Move Wide (16-bit immediate)

These instructions are under **Data-processing (plain binary immediate)**.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>i</td>
<td>1</td>
<td>0</td>
<td>o1</td>
<td>0</td>
<td>0</td>
<td>imm4</td>
<td>0</td>
<td>imm3</td>
<td>Rd</td>
<td>imm8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1 Rn</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>MOV, MOVS (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>MOVT</td>
</tr>
</tbody>
</table>

Saturate, Bitfield

These instructions are under **Data-processing (plain binary immediate)**.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(0)</td>
<td>1</td>
<td>1</td>
<td>op1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op1</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>imm3:imm2</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td></td>
<td>SSAT — logical shift left</td>
</tr>
<tr>
<td>001</td>
<td>!= 00000</td>
<td>SSAT — arithmetic shift right</td>
</tr>
<tr>
<td>001</td>
<td>00000</td>
<td>SSAT16</td>
</tr>
<tr>
<td>010</td>
<td></td>
<td>SBFX</td>
</tr>
<tr>
<td>011</td>
<td>!= 1111</td>
<td>BFI</td>
</tr>
<tr>
<td>011</td>
<td>1111</td>
<td>BFC</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>USAT — logical shift left</td>
</tr>
<tr>
<td>101</td>
<td>!= 00000</td>
<td>USAT — arithmetic shift right</td>
</tr>
<tr>
<td>101</td>
<td>00000</td>
<td>USAT16</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>UBFX</td>
</tr>
</tbody>
</table>
### Advanced SIMD element or structure load/store

These instructions are under **32-bit**.

<table>
<thead>
<tr>
<th>op0</th>
<th>Rn</th>
<th>imm3:imm2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

- **Advanced SIMD load/store multiple structures**
- **Advanced SIMD load single structure to all lanes**

#### Advanced SIMD load/store multiple structures

These instructions are under **Advanced SIMD element or structure load/store**.

<table>
<thead>
<tr>
<th>L</th>
<th>itype</th>
<th>iotype</th>
<th>size</th>
<th>align</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000x</td>
<td>VST4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>VST1</td>
<td>T4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>VST2</td>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>010x</td>
<td>VST3</td>
<td>T3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>VST1</td>
<td>T3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>VST1</td>
<td>T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>100x</td>
<td>VST2</td>
<td>T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>VST1</td>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000x</td>
<td>VLD4</td>
<td>T4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>VLD1</td>
<td>T4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>VLD2</td>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>010x</td>
<td>VLD3</td>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Advanced SIMD load single structure to all lanes

These instructions are under **Advanced SIMD element or structure load/store**.

<table>
<thead>
<tr>
<th>L</th>
<th>itype</th>
<th>iotype</th>
<th>size</th>
<th>align</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>100x</td>
<td>VLD2</td>
<td>T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>VLD1</td>
<td>T2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Advanced SIMD load/store single structure to one lane

These instructions are under Advanced SIMD element or structure load/store.

The following constraints also apply to this encoding: size != 11 && size != 11

Load/store single

These instructions are under 32-bit.
The following constraints also apply to this encoding: op0<1>:op1 != 10

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>!= 1111</td>
<td>000000</td>
<td>Load/store, unsigned (register offset)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>!= 1111</td>
<td>000001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>!= 1111</td>
<td>00001x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>!= 1111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>!= 1111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10x0xx</td>
<td>1111</td>
<td>Load/store, unsigned (immediate, post-indexed)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1100xx</td>
<td>1111</td>
<td>Load/store, unsigned (negative immediate)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1110xx</td>
<td>1111</td>
<td>Load/store, unsigned (unprivileged)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1111</td>
<td>11x1xx</td>
<td>Load/store, unsigned (immediate, pre-indexed)</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>!= 1111</td>
<td>011111</td>
<td>Load/store, unsigned (positive immediate)</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>1111</td>
<td>000000</td>
<td>Load, unsigned (literal)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>000000</td>
<td>Load/store, signed (register offset)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>000001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>00001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>10x0xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>10x1xx</td>
<td>Load/store, signed (immediate, post-indexed)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>1100xx</td>
<td>Load/store, signed (negative immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>1110xx</td>
<td>Load/store, signed (unprivileged)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1111</td>
<td>1111</td>
<td>Load/store, signed (immediate, pre-indexed)</td>
</tr>
<tr>
<td>11</td>
<td>!= 1111</td>
<td>011111</td>
<td>Load/store, signed (positive immediate)</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1111</td>
<td>Load, signed (literal)</td>
<td></td>
</tr>
</tbody>
</table>

### Load/store, unsigned (register offset)

These instructions are under [Load/store single](#).

The following constraints also apply to this encoding: Rn != 1111 & Rn != 1111

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>Rt</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td></td>
<td>STRB (register)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>!= 1111</td>
<td>LDRB (register)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1111</td>
<td>PLD, PLDW (register) — preload read</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td></td>
<td>STRH (register)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>!= 1111</td>
<td>LDRH (register)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1111</td>
<td>PLD, PLDW (register) — preload write</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td></td>
<td>STR (register)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
<td>LDR (register)</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
Load/store, unsigned (immediate, post-indexed)

These instructions are under Load/store single.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>STR (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>LDR (immediate)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Load/store, unsigned (negative immediate)

These instructions are under Load/store single.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>STR (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>LDR (immediate)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Load/store, unsigned (unprivileged)

These instructions are under Load/store single.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>PLD, PLDW (immediate) — preload read</td>
</tr>
<tr>
<td>01</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>PLD, PLDW (immediate) — preload write</td>
</tr>
<tr>
<td>10</td>
<td>STR (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>LDR (immediate)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111
Load/store, unsigned (immediate, pre-indexed)

These instructions are under Load/store single.

The following constraints also apply to this encoding: \( Rn \neq 1111 \land \neg \neg Rn \neq 1111 \)

Load/store, unsigned (positive immediate)

These instructions are under Load/store single.

The following constraints also apply to this encoding: \( Rn \neq 1111 \land \neg \neg Rn \neq 1111 \)
### Load, unsigned (literal)

These instructions are under [Load/store single](#).

- **Decode fields**
  - `size`: 0x
  - `L`: 1
  - `Rt`: 1111
- **Instruction Details**
  - LDR (literal)
  - LDRH (literal)
  - LDRB (literal)

The following constraints also apply to this encoding: `Rn != 1111 && Rn != 1111`

### Load/store, signed (register offset)

These instructions are under [Load/store single](#).

- **Decode fields**
  - `size`: 0x
  - `Rt`: 1
  - `L`: 1
- **Instruction Details**
  - LDRSB (register)
  - LDRSH (register)
  - PLI (register)

The following constraints also apply to this encoding: `Rn != 1111 && Rn != 1111`

### Load/store, signed (immediate, post-indexed)

These instructions are under [Load/store single](#).

- **Decode fields**
  - `size`: 0x
  - `Rt`: 1
  - `L`: 0
- **Instruction Details**
  - LDRSB (immediate)
  - LDRSH (immediate)

The following constraints also apply to this encoding: `Rn != 1111 && Rn != 1111`

### Load/store, signed (negative immediate)

These instructions are under [Load/store single](#).

- **Decode fields**
  - `size`: 0x
  - `Rt`: 1
  - `L`: 0
- **Instruction Details**
  - LDRSB (negative immediate)
  - LDRSH (negative immediate)

The following constraints also apply to this encoding: `Rn != 1111 && Rn != 1111`
Top-level encodings for T32

The following constraints also apply to this encoding: \( Rn \neq 1111 \) & \( Rn \neq 1111 \)

<table>
<thead>
<tr>
<th>Decoded fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 !( \neq ) 1111</td>
<td>LDRSB (immediate)</td>
</tr>
<tr>
<td>00 1111</td>
<td>PLI (immediate, literal)</td>
</tr>
<tr>
<td>01 !( \neq ) 1111</td>
<td>LDRSH (immediate)</td>
</tr>
<tr>
<td>01 1111</td>
<td>Reserved hint, behaves as NOP</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**Load/store, signed (unprivileged)**

These instructions are under **Load/store single**.

<table>
<thead>
<tr>
<th>Decoded fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 !( \neq ) 1111</td>
<td>LDRSBT</td>
</tr>
<tr>
<td>01 !( \neq ) 1111</td>
<td>LDRSHT</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**Load/store, signed (immediate, pre-indexed)**

These instructions are under **Load/store single**.

<table>
<thead>
<tr>
<th>Decoded fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 !( \neq ) 1111</td>
<td>LDRSB (immediate)</td>
</tr>
<tr>
<td>01 !( \neq ) 1111</td>
<td>LDRSH (immediate)</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**Load/store, signed (positive immediate)**

These instructions are under **Load/store single**.

<table>
<thead>
<tr>
<th>Decoded fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 !( \neq ) 1111</td>
<td>LDRSB (immediate)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: \( Rn \neq 1111 \) & \( Rn \neq 1111 \)
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>Rt</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1111</td>
<td><strong>PLI (immediate, literal)</strong></td>
</tr>
<tr>
<td>01</td>
<td>!= 1111</td>
<td><strong>LDRSH (immediate)</strong></td>
</tr>
<tr>
<td>01</td>
<td>1111</td>
<td>Reserved hint, behaves as NOP</td>
</tr>
</tbody>
</table>

### Load, signed (literal)

These instructions are under **Load/store single**.

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>1 1 1 1 1 1 0 0 1 U</th>
<th>size</th>
<th>1 1 1 1 1 1</th>
<th>Rt</th>
<th>imm12</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>Rt</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>!= 1111</td>
<td><strong>LDRSB (literal)</strong></td>
</tr>
<tr>
<td>00</td>
<td>1111</td>
<td><strong>PLI (immediate, literal)</strong></td>
</tr>
<tr>
<td>01</td>
<td>!= 1111</td>
<td><strong>LDRSH (literal)</strong></td>
</tr>
<tr>
<td>01</td>
<td>1111</td>
<td>Reserved hint, behaves as NOP</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Data-processing (register)

These instructions are under **32-bit**.

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>111111010</th>
<th>op0</th>
<th>op1</th>
<th>op2</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1111</td>
<td>0000</td>
<td><strong>MOV, MOVS (register-shifted register) — T2, Flag setting</strong></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>01xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>1xxx</td>
<td><strong>Register extends</strong></td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>0xxx</td>
<td><strong>Parallel add-subtract</strong></td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>10xx</td>
<td><strong>Data-processing (two source registers)</strong></td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>11xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 1111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

### Register extends

These instructions are under **Data-processing (register)**.

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 1 1 1 1 0 1 0 0 0 | op1 U | Rn | 1 1 1 1 | Rd | 1 |(0)rotate | Rm |
|---------------------------------------|-----------------------|-------|-----|-------|-----|----|--------|

### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>U</th>
<th>Rn</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>!= 1111</td>
<td><strong>SXTAH</strong></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1111</td>
<td><strong>SXTH</strong></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>!= 1111</td>
<td><strong>UXTAH</strong></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1111</td>
<td><strong>UXTH</strong></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>!= 1111</td>
<td><strong>SXTAB16</strong></td>
</tr>
</tbody>
</table>
### Parallel add-subtract

These instructions are under [Data-processing (register)](https://example.com).
### Data-processing (two source registers)

These instructions are under [Data-processing (register)](#).

---

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 U H S</td>
<td></td>
</tr>
<tr>
<td>100 1 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101 0 0 0</td>
<td>SSUB16</td>
</tr>
<tr>
<td>101 0 0 1</td>
<td>QSUB16</td>
</tr>
<tr>
<td>101 0 1 0</td>
<td>SHSUB16</td>
</tr>
<tr>
<td>101 0 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101 1 0 0</td>
<td>USUB16</td>
</tr>
<tr>
<td>101 1 0 1</td>
<td>UQSUB16</td>
</tr>
<tr>
<td>101 1 1 0</td>
<td>UHSUB16</td>
</tr>
<tr>
<td>101 1 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110 0 0 0</td>
<td>SSAX</td>
</tr>
<tr>
<td>110 0 0 1</td>
<td>QSAX</td>
</tr>
<tr>
<td>110 0 1 0</td>
<td>SHSAX</td>
</tr>
<tr>
<td>110 0 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110 1 0 0</td>
<td>USAX</td>
</tr>
<tr>
<td>110 1 0 1</td>
<td>UQXSAX</td>
</tr>
<tr>
<td>110 1 1 0</td>
<td>UHSAX</td>
</tr>
<tr>
<td>110 1 1 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 op2</td>
<td></td>
</tr>
<tr>
<td>000 00</td>
<td>QADD</td>
</tr>
<tr>
<td>000 01</td>
<td>QDADD</td>
</tr>
<tr>
<td>000 10</td>
<td>QSUB</td>
</tr>
<tr>
<td>000 11</td>
<td>QDSUB</td>
</tr>
<tr>
<td>001 00</td>
<td>REV</td>
</tr>
<tr>
<td>001 01</td>
<td>REV16</td>
</tr>
<tr>
<td>001 10</td>
<td>RBIT</td>
</tr>
<tr>
<td>001 11</td>
<td>REVSH</td>
</tr>
<tr>
<td>010 00</td>
<td>SEL</td>
</tr>
<tr>
<td>010 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>010 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011 00</td>
<td>CLZ</td>
</tr>
<tr>
<td>011 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100 00</td>
<td>CRC32 — CRC32B</td>
</tr>
<tr>
<td>100 01</td>
<td>CRC32 — CRC32H</td>
</tr>
<tr>
<td>100 10</td>
<td>CRC32 — CRC32W</td>
</tr>
<tr>
<td>100 11</td>
<td>CONSTRAINED UNPREDICTABLE</td>
</tr>
<tr>
<td>101 00</td>
<td>CRC32C — CRC32CB</td>
</tr>
<tr>
<td>101 01</td>
<td>CRC32C — CRC32CH</td>
</tr>
<tr>
<td>101 10</td>
<td>CRC32C — CRC32CW</td>
</tr>
</tbody>
</table>
### Multiply, multiply accumulate, and absolute difference

These instructions are under [32-bit](#).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>11</td>
<td>01</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Multiply and absolute difference

These instructions are under [Multiply, multiply accumulate, and absolute difference](#).

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>01</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>MUL, MULS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>00</td>
<td>MLA, MLAS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>01</td>
<td>MLS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>11</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>10</td>
<td>SMLABB, SMLABT, SMLATB, SMLATT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>01</td>
<td>SMLABB, SMLABT, SMLATB, SMLATT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>10</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>11</td>
<td>SMLABB, SMLABT, SMLATB, SMLATT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>10</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>!≠ 1111</td>
<td>11</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>SMULWB, SMULWT, SMUAD — SMUAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>SMULWB, SMULWT, SMUADX — SMUADX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>SMULWB, SMULWT — SMULWB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>SMULWB, SMULWT — SMULWT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>SMULWB, SMULWT — SMULWT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>SMULWB, SMULWT — SMULWT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#).
Top-level encodings for T32

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 op2 Ra RdLo RdHi</td>
<td>op1 op2</td>
</tr>
<tr>
<td>100 1111 00</td>
<td>SMLSD, SMLSDX — SMLSD</td>
</tr>
<tr>
<td>100 1111 01</td>
<td>SMLSD, SMLSDX — SMLSDX</td>
</tr>
<tr>
<td>100 1111 0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101 1111 00</td>
<td>SMUSD, SMUSDX — SMUSD</td>
</tr>
<tr>
<td>101 1111 01</td>
<td>SMUSD, SMUSDX — SMUSDX</td>
</tr>
<tr>
<td>101 1111 0x</td>
<td>SMMLA, SMMLAR — SMMLA</td>
</tr>
<tr>
<td>101 1111 01</td>
<td>SMMLA, SMMLAR — SMMLAR</td>
</tr>
<tr>
<td>101 1111 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110 1111 00</td>
<td>SMMUL, SMMULR — SMMUL</td>
</tr>
<tr>
<td>110 1111 01</td>
<td>SMMUL, SMMULR — SMMULR</td>
</tr>
<tr>
<td>110 1111 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>111 1111 00</td>
<td>USADA8</td>
</tr>
<tr>
<td>111 1111 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>111 1111 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>111 1111 11</td>
<td>USAD8</td>
</tr>
</tbody>
</table>

These instructions are under 32-bit.

Long multiply and divide

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 op2 RdLo RdHi</td>
<td>op1 op2</td>
</tr>
<tr>
<td>000 0000 01 11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>000 0000 1111 01</td>
<td>SMULL, SMULLS</td>
</tr>
<tr>
<td>001 1111 01 11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>010 0000 0111 01</td>
<td>SDIV</td>
</tr>
<tr>
<td>010 0000 1111 01</td>
<td>UMULL, UMULLS</td>
</tr>
<tr>
<td>011 1111 01 11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100 0000 1001 01</td>
<td>UDIV</td>
</tr>
<tr>
<td>100 0000 01xx 01</td>
<td>SMLAL, SMLALS</td>
</tr>
<tr>
<td>100 0001 01xx 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100 1000 01xx 01</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBB</td>
</tr>
<tr>
<td>100 1001 01xx 01</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBT</td>
</tr>
<tr>
<td>100 1010 01xx 01</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTB</td>
</tr>
<tr>
<td>100 1011 01xx 01</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTT</td>
</tr>
<tr>
<td>100 1100 01xx 01</td>
<td>SMLALD, SMLALDX — SMLALD</td>
</tr>
<tr>
<td>100 1101 01xx 01</td>
<td>SMLALD, SMLALDX — SMLALDX</td>
</tr>
<tr>
<td>100 111x 01xx 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101 0xxx 01xx 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101 10xx 01xx 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101 1100 01xx 01</td>
<td>SMLSLD, SMLSLDX — SMLSLD</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>op1 101</td>
<td>SMLSLD, SMLSLDX — SMLSLDX</td>
</tr>
<tr>
<td>op2 110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op1 110</td>
<td>UMLAL, UMLALS</td>
</tr>
<tr>
<td>op2 0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op1 110</td>
<td>UMAAL</td>
</tr>
<tr>
<td>op2 0111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op1 110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op2 1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op1 111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>